FlashLight:
A Dynamic Detector of Shared State, Race Conditions, and Locking Models in Concurrent Java Programs

THESIS

Scott C. Hale, Captain, USAF

AFIT/GCS/ENG/06-08

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government

AFIT/GCS/ENG/06-08

FlashLight: A Dynamic Detector of Shared State, Race Conditions, and Locking Models in Concurrent Java Programs

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science

Scott C. Hale, B.S.C.S.

Captain, USAF

March 2006

AFIT/GCS/ENG/06-08

FlashLight: A Dynamic Detector of Shared State, Race Conditions, and Locking Models in Concurrent Java Programs

Scott C. Hale, B.S.C.S.

Captain, USAF

Approved:

/signed/	02 Mar 2006
Maj Robert P. Graham, Jr., PhD (Chairman)	date
/signed/	02 Mar 2006
Lt Col T.J. Halloran (Member)	date
/signed/	02 Mar 2006
Dr. Aaron Greenhouse (Member)	date

AFIT/GCS/ENG/06-08

Abstract

Concurrent Java programs are difficult to understand and implement correctly. This difficultly leads to code faults that are the source of many real–world reliability and security problems. Many factors contribute to concurrency faults in Java code; for example, programmers may not understand Java language semantics or, when using a Java library or framework, may not understand that their resulting program is concurrent.

This thesis describes a dynamic analysis approach, implemented in a tool named FlashLight, that detects shared state and possible race conditions within a program. FlashLight illuminates the concurrency within a program for programmers that are wholly or partially "in the dark" about their software's concurrency. FlashLight also works in concert with the Fluid assurance tool to propose Greenhouse-style [8] lock policy models based upon a program's observed locking behavior. After review by a programmer to ensure reasonableness, these models can be verified by the Fluid assurance tool. Our combination of a dynamic tool with a program verification system focused on concurrency fault detection and repair is, to the best of our knowledge, novel and is the primary contribution of this research.

We applied FlashLight to several concurrent Java programs, including a large (∼100kSLOC) commercial web application server. Our case study experiences induced us to improve FlashLight to (1) allow the programmer to specify interesting time *quantums* (e.g., this is the start up phase of my program) and (2) support the common Java programming idiom of not locking shared state during object construction. Both improvements help to reduce false positives. FlashLight introduces an overhead of roughly 1.7 times the original execution time of the program. The most significant limitation of FlashLight is that it is not fully integrated into the Fluid assurance tool with respect to the user experience.

To my wife

Acknowledgements

I would like to thank to my faculty adviser, Maj Robert Graham, for his insight and assistance during this thesis effort. I would also like to show my sincere appreciation to my committee members, Lt Col Timothy Halloran and Dr. Aaron Greenhouse. Their guidance and support throughout the course of this thesis made the effort possible.

Scott C. Hale

Table of Contents

Page

List of Figures

List of Tables

List of Abbreviations

FlashLight:
A Dynamic Detector of Shared State, Race Conditions, and Locking Models in Concurrent Java Programs

I. Introduction

1.1 Troubles With Threads

It is difficult to understand and implement Java concurrency. The query "Java concurrency thread" on *Amazon.com* finds seven textbooks; the same query on the ACM Digital Library finds 200 papers. The sheer number of technical books and papers about Java concurrency testifies to the difficulty of engineering correct concurrent code. Why do we bother with this complexity? Concurrency makes our software more responsive and allows us to take better advantage of available hardware resources. There is a dark side to using concurrency to gain these advantages: concurrent code often has subtle defects that can be maddening to track down and to eliminate. Many factors contribute to defects in concurrent in Java code. Programmers may not understand Java language semantics, or even worse, when using a Java library or framework, may not understand that their code is concurrent. Regardless of the cause, faults in concurrent code can lead to race conditions: anomalous behavior due to an unexpected program dependence on the relative timing of events. Avoiding race conditions by holding locks during critical sections of code can unfortunately lead to deadlock: a situation where two or more threads are unable to proceed because each is waiting for one of the others to release a resource.[1] These defects are difficult to track down because they are effectively nondeterministic.

The Fluid project[2] is dedicated to developing techniques that change this situation in a positive manner. This project includes researchers at Carnegie Mellon

[1] Our definitions for *race condition* and *deadlock* are adapted to the Java programming language from the definitions at `http://onlinedictionary.datasegment.com`.

[2] `http://www.fluid.cs.cmu.edu`

University, the Air Force Institute of Technology, and the University of Milwaukee–Wisconsin. The Fluid assurance[3] tool is an Eclipse-based tool focused on the practical verification of mechanical (non-functional) design intent about Java code. This specification focus differs from the traditional focus in much of the program verification literature on *functional* properties—models of component input/output behavior. Germane to our work, the Fluid assurance tool supports the specification and verification of how locks protect state within a Java program, which we refer to as a *lock policy*. This technique, developed by Greenhouse [8, 9], has proved successful in uncovering and correcting defects in open source, commercial, and governmental software systems. The technique has also been judged practical and adoptable by practicing programmers during several on-site case studies with commercial software companies and Government organizations.

Our work, the development of the FlashLight tool to illuminate the concurrency within a program, is a direct result of the observation that programmers are sometimes "in the dark" about their software's concurrency. This observation was made by members of the Fluid project, to some degree, during all of the on-site case studies, but was the most noticeable (as described below) during a Government on-site case study.[4]

1.1.1 In the Dark. A troubling problem encountered during a Government on-site case study was that programmers did not realize that significant portions of their code were, in fact, concurrent. This made it difficult for them to gain value from the Fluid assurance tool (in terms of defects identified and fixed) because the tool requires the programmer to express lock policy models for it to verify. To help the programmer get started, the tool scans the code and highlights concurrent constructs within the code, e.g., threads being started or locks being acquired and subsequently

[3]We use the word *assurance* as a synonym for *verification*—proof that an implementation is consistent with a precise behavioral specification or model.

[4]Personal communication with members of the Fluid project who participated in the on-site case studies of commercial and governmental software systems.

released, for the programmer to examine. The intent is to signpost possible locations in the code where expressing a lock policy model might be possible. We found, however, that in code written by programmers "in the dark" about the concurrency within their software, these static "signposts" to guide lock policy expression did not exist. We posit, based upon informal discussions with the programmers participating in the case study, two possible reasons:

- The concurrency was imposed by a third-party library (e.g., Swing) or a separately developed component and the programmer lacked an understanding of the concurrency introduced by the library or component into his or her code.
- The programmer held the misconception that the Java language semantics automatically ensure race-free code.

We believe the problem of programmers being "in the dark" about concurrency is more widespread in practice than one might at first believe. This opinion is based upon our observation that these Java programmers are, in other respects, competent and hardworking professionals and that the software systems they develop and maintain are considered mission critical to the Government organization that operates them.

1.2 This Thesis

This thesis describes a dynamic analysis approach, implemented in a tool named FlashLight, that detects shared state and possible race conditions within a program. Based upon the program's observed locking behavior, the tool also proposes Greenhouse-style [8] lock policy models that can, after review by a programmer to ensure reasonableness, be assured by the Fluid assurance tool. FlashLight is designed to be synergistic with the Fluid assurance tool: it is another step toward the goal of improving the quality of large real-world software system in a practical manner.

The combination of a dynamic tool with a static program verification system focused on concurrency fault detection and repair is, to the best of our knowledge,

novel and is the primary contribution of this research. A secondary contribution of this work is the extension of the lock-set analysis algorithm (discussed in Chapter II) to use what we call *quantums*. Quantums allow the programmer to specify one or more "interesting" periods of time during a program's execution. For example, quantums can be used to identify the "start up," "steady state," and "shut down" phases of a program's execution. Quantums allow the programmer to "focus" the tool on particular periods of the program's execution which may suffer from intermittent failure or be poorly understood.

1.3 Tool Use Overview

FlashLight instruments Java programs, monitors their execution by collecting data about field use and held locks, and aggregates the run-time data to produce reports for the programmer to examine. A programmer using FlashLight repeatedly follows this process:

1. *Customize the instrumentation.* The programmer provides the tool with information about his or her program. Specifically, the programmer notes when the analysis should start and stop collecting data. Optionally, any quantums of time he or she wishes to distinguish are specified. The programmer may also restrict data collection to a subset of the program's classes. Finally, based upon these specifications, the programmer lets FlashLight weave required instrumentation into their program.
2. *Run the program.* The programmer invokes a large test suite or puts the program into any "production-like" situation he or she deems of interest. The goal is to stimulate the execution of as many dynamic paths within the program as possible so that FlashLight can produce the best possible results for the programmer. FlashLight collects data as the program runs and creates several XML files when the program exits.

3. *Examine the reports.* FlashLight produces a suite of web pages that the programmer can now examine to better understand the concurrency in his or her program.

As is typical in almost any dynamic analysis, FlashLight only "sees" a subset of all possible program execution paths. Its results are, therefore, *incomplete*. In terms of reported shared state, the tool is *sound*, because the identification of shared state does not require any understanding of the program's functionality. Race condition detection by FlashLight is *unsound*. This is because determination of a race condition with respect to the semantics of the application depends upon having higher level, application-specific semantic information that FlashLight lacks. Put another way, FlashLight has no idea what the program's intended functionality is, so it can't be sure if an observed interaction between threads is a race condition or programmer intended behavior.

FlashLight uses the quantum specification provided by the programmer as a surrogate for more detailed program design intent. FlashLight's use of such coarse design intent is intentional because any design intent we elicit from the programmer has an expression cost. Asking a programmer on a deadline to pay too high of a cost, in terms of their time, can cause the tool to be impractical.

1.4 Motivating Example: A "Maze" of Concurrency

As we have noted above, the primary hypothesis of this research is that programmers do not always fully understand the concurrency of their programs. In the example we now present, the Swing library imposes concurrency upon an apparently single-threaded program, Maze ADT.

The Maze ADT program is used at AFIT to instruct students about data structures and algorithms. The application has a graphical user interface (GUI) shown in Figure 1.1 that is constructed using the Swing library.

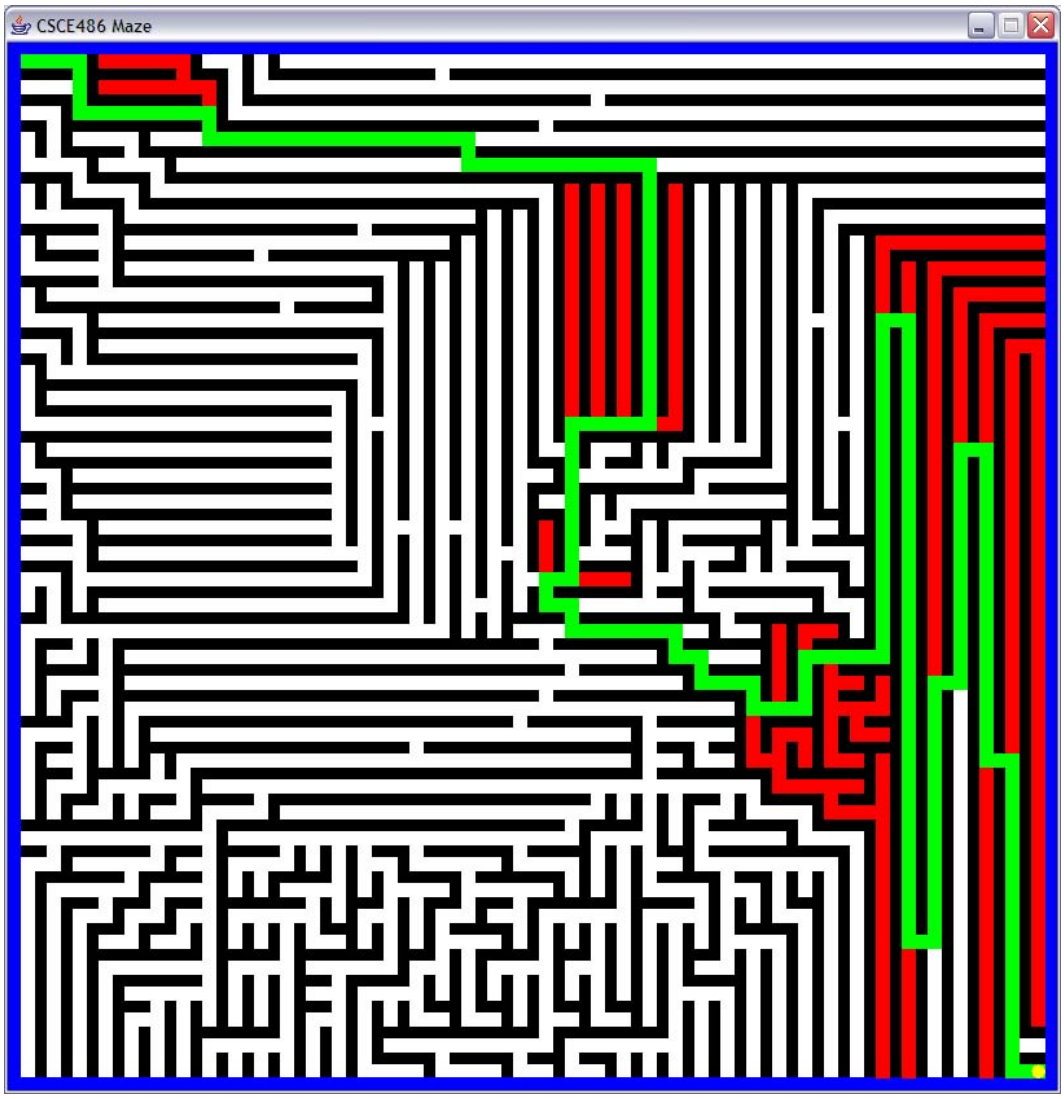


Figure 1.1: The Maze ADT User Interface. The Maze ADT program used to demonstrate algorithms for solving random mazes. Despite the use of double-buffering, the original program appears to draw the path chosen by the algorithm in "fits and starts." This visual artifact is a symptom of the race condition in the original program code shown in Figure 1.3.

```
Exception in thread "AWT-EventQueue-0"
java.util.ConcurrentModificationException
    at java.util.LinkedList$ListItr.checkForComodification(Unknown Source)
    at java.util.LinkedList$ListItr.next(Unknown Source)
    at Maze.drawEntirePath(Maze.java:35)
    at Maze.paint(Maze.java:27)
        at ...
    at sun.awt.RepaintArea.updateComponent(Unknown Source)
```

Figure 1.2: Concurrent Modification Error generated by the Maze. During execution of the original Maze ADT program, thousands of exceptions exactly like this one are output to the console. We have modified the line number references so that they correspond to the code shown in Figure 1.3.

1.4.1 Darkness. Consider the elided source code for the Maze ADT program shown in Figure 1.3. The primary data structure of the application is the `LinkedList pointList` which stores a list of `Point` objects. Each `Point` object has an associated color. The color depends on whether the `Point` exists on the potential solution path, on a dead end, or on a path not yet checked by the algorithm trying to solve the maze. Because the programmer thought the application was single-threaded, there is no synchronization, or locking, in the code.

Despite the use of double-buffering, the program appears to draw the path chosen by the maze solving algorithm in "fits and starts." This visual artifact is a symptom of a Swing-imposed race condition in the original program. Another symptom of the race condition in the program is the thousands of exceptions exactly like the example shown in Figure 1.2 that appear on the console. These symptoms brought the programmer to us for help. The programmer realized that his program with "no concurrency" probably had some concurrency that he "didn't put into it"—primarily due to the stream of `ConcurrentModificationException` exceptions produced by his program. This exception is an artifact of the "fail-fast" design of the Java collections classes. It is interesting that if the field `pointList` did not use the "fail-fast" Java collection class `LinkedList` (e.g., it used an array), the programmer might never have noticed the concurrency fault in his program.

```
1 public class Maze extends JFrame {
2
3   private final LinkedList<Point> pointList;
4
5   ...
6
7   public Maze(String mazeTitle, int Cell_Size, int Wall_Size, ...) {
8     ...
9     pointList = new LinkedList<Point>();
10    ...
11  }
12
13  public void addPointToPath(int x, int y, Color c) {
14    ...
15    Point point = new Point(x, y, c);
16    pointList.add(point);
17    this.repaint();
18  }
19
20  public void changeTopColor(Color c) {
21    Point point = pointList.getLast();
22    point.c = c;
23  }
24
25  @Override public void paint(Graphics g) {
26    ...
27    drawEntirePath(g);
28  }
29
30  private void drawEntirePath(Graphics g) {
31    Iterator<Point> i = pointList.iterator();
32    if (i.hasNext()) {
33      Point lastPoint = drawSquare(g, i.next());
34      while (i.hasNext())
35        lastPoint = drawSquareTo(g, lastPoint, i.next());
36    }
37  }
38
39  private Point drawSquare(Graphics g, Point p1, Point p2) {
40    g.setColor(p1.c);
41    ...
42  }
43 }
```

Figure 1.3: An elided version of the original `Maze` class which contains a subtle race condition on the contents of `pointList` due to its use of Swing.

```
1 pointcut steadyState() : call(setVisible(..));
2
3 after() : steadyState()  {
4   advanceQuantumWithCollection("Steady State");
5 }
```

Figure 1.4: The definition of a quantum for the Maze ADT program that instructs FlashLight to begin dynamic analysis when the GUI is made visible with a call to the `setVisible` method and to end when the program exits.

- **Field c in class Point**

Instance	Thread Name	Read Count	Writes Count	Locks Held by Thread
Maze.3341135				
	AWT-EventQueue-0	7652	0	■ No lock is held at this field access
	main	0	2368	■ No lock is held at this field access

Figure 1.5: FlashLight detected that two threads access the field `c` in class `Point`. The accesses occur in an instance of the `Maze` class. FlashLight classifies the accesses as a potential race because two threads accessed the field without holding a common lock.

1.4.2 Shining the FlashLight. We now use FlashLight to help "shed some light" on the erroneous concurrency of our program. We assume the problem occurs after the GUI is visible, based on the symptoms described above, such as the visual artifact of the path being drawn in "fits and starts." Therefore, our first step is to configure FlashLight instrumentation with one quantum that begins when the GUI is made visible and ends when the program exits. The definition of this quantum focuses FlashLight on what we might call the program's "steady state" phase of execution. The definition of this quantum is shown in Figure 1.4. Quantums are specified using AspectJ syntax (AspectJ is described later). The code in Figure 1.4 captures calls to the method `setVisible`. When a call occurs, a new quantum is created labeled `Steady State`. This new quantum stores all the data captured by the instrumentation. Upon completing the maze, the quantum's data is analyzed to determine if any state is shared among threads.

After we finish our quantum definition, Maze ADT is compiled with the AspectJ compiler to "weave" in required instrumentation. In addition, the FlashLight JAR (which contains code to store, analyze, and output results) is added to the program's classpath. At this point the program is executed. FlashLight causes the program to output several results files that can be opened in a web browser.

A portion of FlashLight's output is shown in Figure 1.5. Because of our quantum configuration, only one field is highlighted in the output. The field `c` from the

```
Thread AWT-EventQueue-0 Read Count = 7652 Write Count = 0
  Reads Stack Trace
        at Maze.drawSquare(Maze.java:40)
        at Maze.drawEntirePath(Maze.java:33)
        at Maze.paint(Maze.java:27)
        ...
```

Figure 1.6: FlashLight output showing the stack trace for thread `AWT-EventQueue-0`'s access of the field `c` in the `drawSquare` method at line 40 in Figure 1.3.

```
Thread main Read Count = 0 Write Count = 2368
  Writes Stack Traces
         at Maze.changeTopColor(Maze.java:22)
         ...
```

Figure 1.7: FlashLight output showing the stack trace for thread `main`'s access of the field `c` in the `changeTopColor` method at line 22 in Figure 1.3.

`Point` class is reported as shared state and as a potential race condition. The results in Figure 1.5 report that this field is accessed by two threads: the `main` thread, which the programmer expected, and `AWT-EventQueue-0`, which is a surprise to the programmer!

In this example, FlashLight clearly points the programmer in the direction of the program fault. It can't, however, fix a muddled design for the programmer. The output contains additional information to assist the programmer in the form of stack traces. Figures 1.6 and 1.7 show the stack traces for the threads `AWT-EventQueue-0` and `main` respectively. After examining all the FlashLight output, the programmer can determine that the `Point` object instances being shared are all contained within the `pointList` field of a single `Maze` object instance (declared at line 9 of Figure 1.3). The tool output has only identified the `c` field of `Point` object instances as being shared. However, this is an artifact of the current implementation—the programmer realizes that, in fact, the entire state of each `Point` object instance might (perhaps due to future code changes) be shared. Further, based upon the locality of the accesses within the `Maze` class, the programmer realizes that only `Point` object instances contained in `pointList` are being concurrently accessed.

The programmer must also study the Swing library documentation to understand the genesis of the AWT event queue thread and why calls to the `Maze` object's `paint` method are made by that thread and not the `main` thread as the programmer expected.

1.4.3 Eliminating the Race. Clearly we need to protect the `c` field of `Point` from being accessed concurrently. We see three uses of the field in Figure 1.3 at lines 15, 22, and 40. These field accesses lead to the concurrent modification of the `pointList` data structure. The concurrent modification occurs because while the `Maze` object's `paint` method is called by the `AWT-EventQueue-0` thread, the `addPointToPath` and `changeTopColor` methods are called by the program's `main` thread, which triggers the "fail-fast" exceptions from the `LinkedList pointList`. This explains the stream of `ConcurrentModificationException` exceptions, but not why our tool did not note the concurrent access to the internals of the shared `LinkedList` implementation. This highlights a limitation of our tool: FlashLight can only detect shared state within code it has instrumented. In this example the programmer did not use the AspectJ compiler to instrument the SDK libraries (typically in a file named `rt.jar`) that contain the code for the `LinkedList` class. Only the programmer's own code was instrumented. This is why FlashLight discovered `c` to be shared state and missed the shared internals of the `LinkedList pointList`.

Our programmer attempts to correct the fault by synchronizing each method that accesses the `pointList` data structure: `addPointToPath`, `changeTopColor`, and `drawEntirePath`. Note that his implicit design intent is that access to the contents of the `pointList` should be protected by a lock on the enclosing `Maze` object. Does this really fix the program fault? The programmer has high hopes, but wants to be sure. He would like to verify this lock policy using the Fluid assurance tool. Hence, he runs FlashLight again using the same configuration to have it propose a Fluid annotation.

1.4.4 FlashLight Proposes a Lock Policy. With the synchronization in place, the race condition symptoms described above disappear during the execution of the

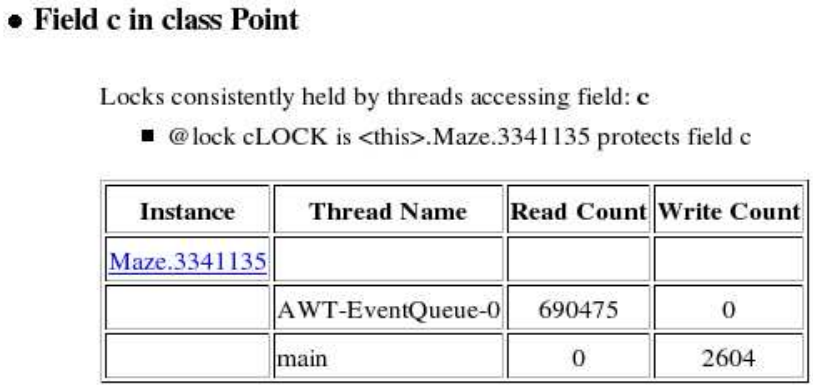


Figure 1.8: With synchronization in place, field `c` is consistently protected by the `Maze` object. FlashLight reports the field is protected using a Fluid @lock promise.

program. The programmer is optimistic, but knows that a single execution of the program is not a sound assurance that a race condition has really been fixed. When the programmer reviews the FlashLight output he notes that the field `c` is still detected as shared state. However, this time FlashLight reports that, at least for the particular run of the program it observed, `c` is consistently protected by a lock on a `Maze` object. The actual FlashLight output is shown in Figure 1.8.

As seen in Figure 1.8, FlashLight proposes a lock policy model in a syntax similar to the Fluid `@lock` annotation. The proposed model in this case is

@lock cLOCK is <this>.Maze.3341135 protects field c

which indicates that locking a Maze object should protect the field `c` of `Point` objects. Again, FlashLight points the programmer in the right direction, but can't divine design intent. Some thought is still needed to express the correct Fluid annotations to assure the programmer's fix is correct.

1.4.5 Verifying the Lock Policy. Armed with the proposed locking model provided by FlashLight, it is possible to add Fluid annotations, called *promises*, to the code. Using these annotations, the Fluid assurance tool can verify that our program no longer contains the race condition. The Fluid assurance tool, unlike FlashLight,

```
1 /**
2  * @region MazeRegion
3  * @lock MazeLock is this protects MazeRegion
4  */
5 public class Maze extends JFrame {
6   /**
7    * @unshared
8    * @aggregate Instance into MazeRegion
9    */
10  private final LinkedList<Point> pointList;
11  /**
12   * @singleThreaded
13   * @starts nothing
14   */
15  public Maze(String mazeTitle, int Cell_Size, int Wall_Size, ...) {
16    ...
17    pointList = new LinkedList<Point>();
18    ...
19  }
20
21  public synchronized void addPointToPath(int x, int y, Color c) {
22    ...
23    Point point = new Point(x, y, c);
24    pointList.add(point);
25    this.repaint();
26  }
27
28  public synchronized void changeTopColor(Color c) {
29    Point point = pointList.getLast();
30    point.c = c;
31  }
32
33  @Override public void paint(Graphics g) {
34    ...
35    drawEntirePath(g);
36  }
37
38  private synchronized void drawEntirePath(Graphics g) {
39    Iterator<Point> i = pointList.iterator();
40    if (i.hasNext()) {
41      Point lastPoint = drawSquare(g, i.next());
42      while (i.hasNext())
43        lastPoint = drawSquareTo(g, lastPoint, i.next());
44    }
45  }
46
47  private Point drawSquare(Graphics g, Point p1, Point p2) {
48    g.setColor(p1.c);
49    ...
50  }
51 }
```

Figure 1.9: The corrected `Maze` class (changes from Figure 1.3 are italicized) with Fluid promises added to precisely specify its lock policy: when accessing the contents of `pointList` a lock on the object instance (i.e., `this`) must be held. The Fluid assurance tool verifies this lock policy is consistent with the code.

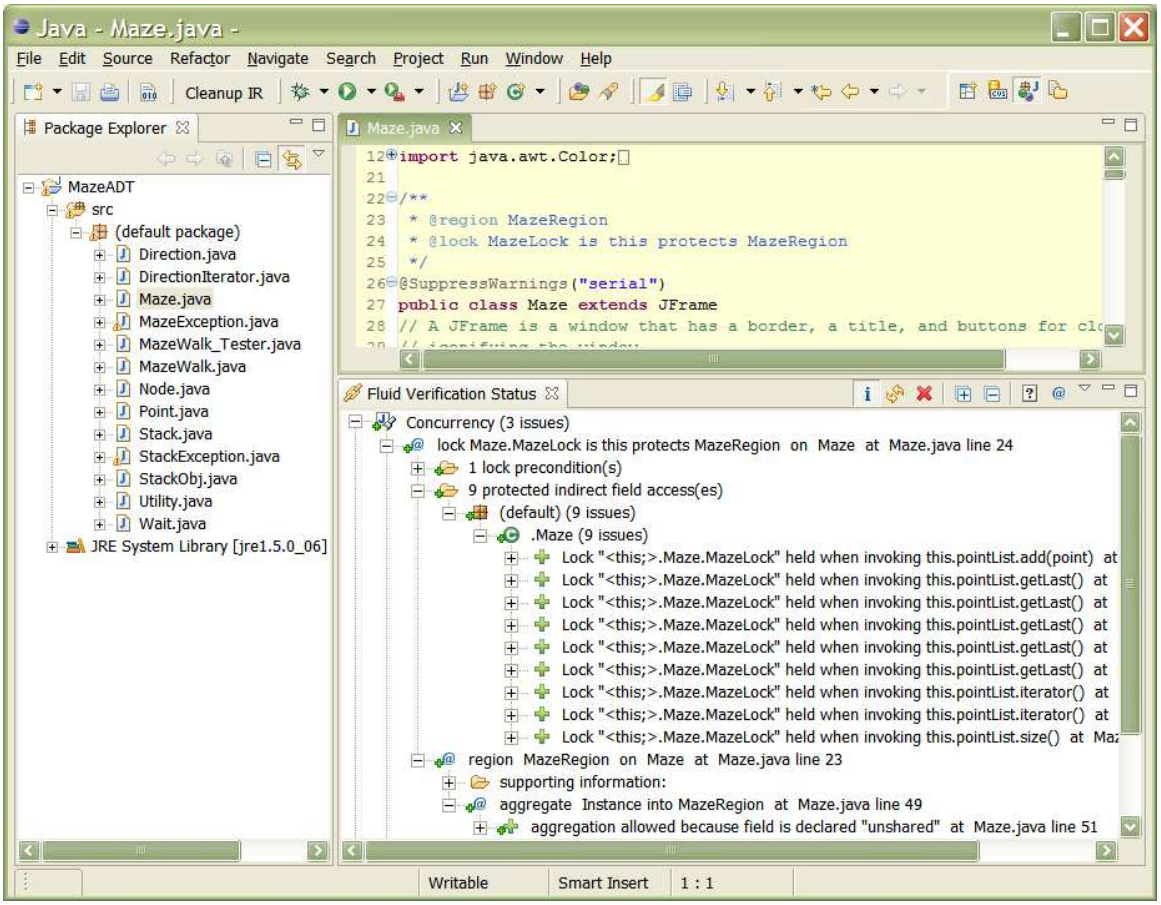


Figure 1.10: The Fluid Assurance tool (running inside the Eclipse Java IDE) applied to corrected Maze class. The tool is able to assure the code is consistent with the locking policy. The "Fluid Verification Status" display at the bottom-right indicates model–code consistency via the green plus icon prefixing the second line of its results.

considers all possible paths the program may take at runtime, and therefore its results are sound.

Recall that the implicit design intent behind the fix made to the Maze ADT code is that access to the contents of the `pointList` is protected by using the enclosing `Maze` object as a lock. Figure 1.9 shows the corrected Maze class (i.e., `synchronized` has been added to all needed methods) annotated with necessary Fluid promises to assure its lock policy. The results, which indicate that the Maze ADT code is consistent with the annotated lock policy, are shown in Figure 1.10. Understanding the details of the promises in Figure 1.9 and the details of the verification results produced by the Fluid assurance tool in Figure 1.10 is beyond the scope of this thesis; however, we refer the interested reader to [8] and the "Introduction to Declaring Design Intent in Fluid" on the Fluid project web site.[5]

1.5 Case Studies

We applied FlashLight to several concurrent Java programs including educational software, an established open source project, and a commercial system. These case study experiences motivated improvements to FlashLight:

- We reduced the number of false positives in the output by improving the lock-set algorithm used by the tool to support common Java programming practices.
- We continuously improved the format and contents of the reports produced by the tool to increase their usefulness and comprehensibility.
- We discovered and repaired several serious flaws in the tool.

As part of our case study, we also evaluated the overhead incurred by using FlashLight. During our trials, the open source text editor jEdit took approximately 1.7 times longer to execute while being inspected with FlashLight. During our commer-

[5] `http://www.fluid.cs.cmu.edu:8080/Fluid/annotation-handout.html`

cial case study, the commercial programmers noted no significant difference in the performance of their application server except for an increase in memory use.

1.6 Outline

The remainder of this document is organized as follows:

- Chapter II, "Definitions and Prior Work," provides precise formal definitions for shared state, race condition, and what we mean by consistent and inconsistent protection of state in a concurrent program. This chapter also frames our work in the context of prior research.
- Chapter III, "Tool Use," describes details of how to use FlashLight.
- Chapter IV, "Tool Engineering," describes the design and implementation of FlashLight. This chapter describes our approach to limiting false positive results reported by the lock-set detection algorithm used by FlashLight. It also describes our approach to proposing lock models usable by the Fluid assurance tool.
- Chapter V, "Case Studies," describes several case studies, one with a top-10 business software company, to which we applied our FlashLight prototype tool. This chapter reports the strengths and weaknesses of FlashLight found on these case studies.
- Chapter VI, "Conclusion," summarizes our results and covers possible future work.

II. Definitions and Prior Work

This chapter discusses relevant prior work in the area of analysis techniques and tools for understanding concurrent programs. We focus on dynamic analysis techniques for race condition detection because this is the focus of FlashLight, but we also note tools based upon model checking or static analysis. Furthermore, we use this chapter to precisely define several terms and provide a quick introduction to aspect-oriented programming (AOP), which FlashLight uses to instrument programs.

Section 2.1 defines shared state, race condition, and what we mean by consistent and inconsistent protection of state in a concurrent program. Section 2.2 discusses three approaches for dynamically identifying possible race conditions: happens-before, lock-set, and the O'Callahan–Choi hybrid. We also discuss why we chose the lock-set approach for FlashLight. Sections 2.3 and 2.4 review related work using model checking and static analysis, respectively. Section 2.5 describes aspect-oriented programming and reviews prior dynamic analysis tools, similar to FlashLight, that have used this technology to instrument programs.

2.1 Definitions

In this next section, we define shared state in a concurrent Java program and formalize the notion of a race condition.

2.1.1 What is Shared State? Java programs typically have more than one thread of execution. Each thread of execution has its own stack, but threads share a single heap, so all objects are available to all threads. It is this reason that all fields, instance and static, are available to be shared. For the Java programming language, we define shared state as all the fields accessed by multiple threads. By design, fields are the only possible shared state within a Java program [7].[1] It is not possible to communicate across threads of execution via local variables or parameters

[1] We note, for the sake of completeness, that Java threads may communicate via pipes. However, we do not consider pipes to be difficult for programmers to identify in a concurrent program and, therefore, do not consider them further in this work. For more information on pipes see [7, 22].

(which exist as part of a single thread's stack). Not all state within a concurrent Java program is shared. For example, particular object instance fields or static fields may in actuality be accessed by a single thread only.

Choi, et al. in [3] propose a formalization for access events that occur within one execution of a particular program. We use this formalism to precisely define our notion of shared state, inconsistently protected shared state, and consistently protected shared state. Choi, et al. define an *access event* to consist of a 5-tuple (m, t, L, a, s), where

- m is the memory location accessed
- t is the thread which performs the access
- L is the set of locks held by t at the time of the access
- a is the access type $\{\mathsf{READ}, \mathsf{WRITE}\}$
- s is the source location of the access instruction

The source reference, s, is only used for reporting information about events. A program execution defines a set of access events, E.

We can use this formalism to precisely describe the shared state of a Java program. For this purpose, m is restricted to be the location of a field inside an object in the program's heap. Thus, the set of shared state within a program, S_{shared}, is defined as

$$S_{shared} = \{m \mid \forall e_x, e_y (e_x \in E \wedge e_y \in E \wedge \text{shared}(e_x, e_y) \wedge m = e_x.m)\}$$

where the predicate indicating a shared access is defined as

$$\text{shared}(e_1, e_2) \ : \ \begin{array}{c} e_1.m = e_2.m \ \wedge \ e_1.t \neq e_2.t \ \wedge \\ (e_1.a = \mathsf{WRITE} \ \vee \ e_2.a = \mathsf{WRITE}) \end{array}$$

for any two access events e_1 and e_2. Informally, a shared access occurs any time a field is accessed by more than one thread and at least one access is a WRITE . Our definition of shared state does not consider if any locks are held when the field is accessed.

2.1.2 What is a Race Condition? We have informally defined a race condition as anomalous program behavior due to an unexpected critical dependence on the relative timing of events. In this section we make this definition more precise.

Using the access event formalism described above, we adopt the definition of Choi, et al. in [3] for a potential race condition. Given two access events, e_1 and e_2, a potential race condition can be defined as the predicate

$$\text{race}(e_1, e_2) \ : \ \ \text{shared}(e_1, e_2) \ \wedge \ e_1.L \ \cap \ e_2.L = \emptyset$$

and the set of state with the potential for a race condition, S_{race}, is defined as

$$S_{race} = \{m \mid \forall e_x, e_y (e_x \in E \wedge e_y \in E \wedge \text{race}(e_x, e_y) \wedge m = e_x.m)\}.$$

Note that S_{race} is the set of all shared state that is inconsistently protected or not protected at all. State within this set creates the potential for a race condition within the program; however, it is not possible to conclude that this necessarily indicates a program fault. Why? Because a policy of non-lock single-threaded access may exist within the program that serves to ensure a race condition does not occur. We may conclude, however, that any state in S_{race} is suspicious and should be considered "guilty until proven innocent" in terms of creating the potential for a race condition.

These definitions are the basis for the detection of shared state and possible race conditions in FlashLight. FlashLight extends the above notion of E to create multiple sets of access events throughout the lifetime of the program's execution. A programmer-specified subset of E is called a *quantum*—a partition of the program

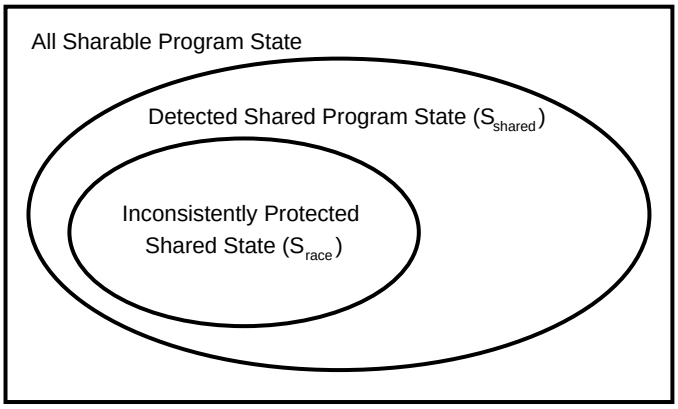


Figure 2.1: A diagram illustrating the relationship between all sharable program state within a particular execution of a program (i.e., Java fields within object instances), state that was shared by one or more threads, S_{shared}, and shared state that was inconsistently protected by locks, S_{race}. The inconsistent protection of the state in S_{race} could indicate the potential for a race condition on that state.

execution time (e.g., startup, steady state, shutdown). It is within a particular time quantum that FlashLight searches for shared state and potential race conditions.

This definition implies that all state that is inconsistently protected is also shared state, but the reverse does not hold. Therefore, $S_{race} \subseteq S_{shared}$, as is shown in Figure 2.1. Finally, we emphasize that because S_{shared} and S_{race} are constructed from data from a single execution of the program, these sets are incomplete. State that, in fact, is shared might not appear in S_{shared} because it was not shared in that particular execution of the program. State that is, in fact, inconsistently protected within the program might not appear in S_{race} because it was consistently protected in that particular execution of the program.

Consider the set, $S_{prot} = S_{shared} \setminus S_{race}$, i.e., the set of shared state that is consistently protected by the same set of locks. The set of locks protecting some state, m, may be defined as

$$\text{locks}(m) = \bigcap_{e \in \{x \in E \,|\, x.m = m\}} e.L$$

where if $m \in S_{race}$ then it will always be the case that locks(m) = $\emptyset$. S_{prot} is, like S_{shared} and S_{race}, incomplete.

2.1.3 Java Mapping. FlashLight is a tool to analyze Java. We now relate the above access event formalism to the Java language.

- m: A memory location. In Java, m references an object instance on the heap or fields within an object instance on the heap.
- t: A thread. In Java, t refers to a Java thread.
- L: A set of held locks. In Java, a single lock is associated with every object, array, and class. L is the set of locks held by the Java thread which accessed m.
- a: Either READ or WRITE depending upon the type of access to m.
- s: For Java we can track not only the compilation unit (i.e., Java file) and line number of the access event, but also the stack trace leading up to the access event.

2.2 Dynamic Analysis Race Condition Detection Algorithms

Dynamic analyses for detecting race conditions are typically classified as *on-the-fly* or *post-mortem* which classifies when these analyses produce their results. FlashLight is a post-mortem detector.

On-the-fly detectors collect run-time information about a program and report errors as they occur. Schonberg describes an on-the-fly detector in [21] and argues that the biggest advantage for this type of detector is system resource preservation. An on-the-fly tool discards information when it becomes apparent the information is no longer needed. For example, when a race condition is found and reported, the accompanying trace information is disposed. System resource consumption, especially memory, is a valid concern: in FlashLight we only keep unique stack traces. Each stack trace has an associated counter. If we encounter multiple instances of the same trace, we increment the counter instead of storing multiple instances of the stack

trace. However, FlashLight is a post-mortem detector and we do use a significant amount of program memory to store analysis data.

Post-mortem detectors evaluate information collected (and saved) during one or more runs of a program for potential race conditions. Because FlashLight is a post-mortem detector, we focus on prior work using this approach. We describe three post-mortem techniques used to dynamically detect race conditions. Table 2.1 summarizes the positive and negative aspects of three dynamic race condition detectors described in the literature. FlashLight implements the lock-set technique that compares the set of locks held by each thread at a given access event to determine if state is consistently protected. We chose the lock-set approach because of its straightforward engineering and its ability to be extended to support time quantums.

Program analyses are susceptible to two kinds of errors with respect to the results they report: *false positives* and *false negatives*. A false positive result is when the analysis reports a result that, in fact, is not really a result. For example, if an analysis reports that concurrent access to a field is a race condition, but it turns out that the programmer intended the observed concurrent access (for some reason), then the program was correct (with respect to its programmer intended functionality) and the analysis has produced a false positive result. Here we say that the analysis is being *conservative*. A false negative result is when the tool does not report a result that, in reality, exists in the program. For example, if a program contains a race condition that is not reported by an analysis, then the analysis has produced a false negative result. Here we say that the analysis is being *gullible*.

Another measure used to compare dynamic analysis approaches is *overhead*. Because the analysis runs "together" (in our case on the same Java Virtual Machine (JVM)) with the target program, the analysis utilizes additional system resources (e.g., memory and time). We define the term *overhead* as the additional resources required to execute both the target program and the dynamic analysis. A large over-

Table 2.1: Positive and negative aspects of post-mortem dynamic analysis race condition detection algorithms in the literature. This comparison guided our selection of the lock-set algorithm for FlashLight.

Technique	Pros / Cons
Happens-before [14]	Pro: No false positive results
	Con: False negative results (i.e., gullible)
	Con: High runtime overhead (i.e., slows program)
Lock-set [20]	Pro: No false negative results
	Pro: Less runtime overhead than happens-before
	Pro: Simple algorithm
	Con: False positive results (i.e., conservative)
O'Callahan–Choi Hybrid [17]	Pro: Improved precision over other techniques
	Pro: Less runtime overhead than happens-before
	Con: Complex Algorithm
	Con: False positives from lost lock acquisitions
	Con: False negatives from lost memory acquisitions

head equates to requiring more system resources to execute both the target program and the dynamic analysis.

We now describe each of the three dynamic analysis approaches to detecting race conditions summarized in Table 2.1 and contrast them to FlashLight.

2.2.1 Happens-Before. The happens-before ordering is a partial order on all the events of all the threads in a concurrent execution of a program. This ordering was introduced by Lamport in [14] to describe the order of events based on known or deduced information. Given a single thread, the events are ordered in the order in which they occur. Given multiple threads, events are ordered based on the properties of the synchronization objects they access.

O'Callahan and Choi argue in [17] that happens-before produces no false positives because for every event the happens-before detection finds, there exists a thread scheduling where the threads in question could execute "simultaneously" and therefore produce a race condition. Based solely on this analysis, one might assume that the majority of the dynamic analysis tools would implement happens-before detection

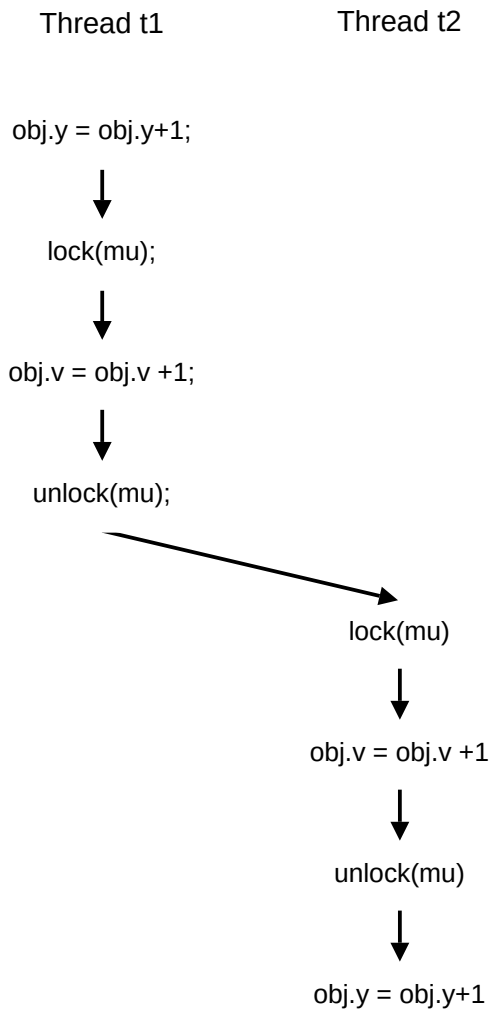


Figure 2.2: This program contains a race condition on y, but the fault will not be reported by a happens-before detector that observes this particular execution interleaving (a false negative). Both threads access memory location y in an unprotected fashion (a race condition); however, a happens-before race condition detector does not detect the race because in this sequence of events, thread t1 holds the lock (mu) before thread t2, so the accesses to y are ordered in this interleaving.

to uncover race conditions. This is not the case for two reasons: (1) A happens-before detector has a high runtime overhead. The best implementation to date, TRaDe described by Christiaens and DeBosschere in [4], slows Java programs by roughly a factor of five. (2) A happens-before detector can produce false negatives, i.e., it can fail to detect potential race conditions that were dynamically observed. Figure 2.2 demonstrates a race condition missed by happens-before (a false negative). Two threads execute code to manipulate fields v and y of an object instance referenced by obj. The field v is protected from concurrent access by locking on mu. However, the field y has no synchronization. The program has a potential race condition on y that is missed by the happens-before detector because in this sequence of events, thread t1 holds the lock mu before thread t2, so the accesses to y are ordered in this particular interleaving. A happens-before based tool would only find this error if the scheduler executes thread t2 before thread t1 [20].

2.2.2 Lock-Set. A lock-set detection algorithm compares the locks held by threads when they access state. If inconsistent sets of locks are used when accessing state, a potential race condition is reported. FlashLight uses lock-set detection augmented with time quantums.

We describe the lock-set algorithm used by the Eraser application [20]. The premise of lock-set analysis is that every shared field access is protected by a lock. O'Callahan and Choi in [17] formalize this with their lock-set hypothesis.

> Whenever two different threads access a shared data memory location, and one of the accesses is a write, the two accesses are performed holding some common lock

This hypothesis is the basis for determining which field accesses produce race conditions in lock-set.

Savage, et al. in [20] introduce the lock-set dynamic analysis algorithm via their Eraser tool. The lock-set algorithm maintains a set of candidate locks $C(m)$ for each shared field m. This set contains the locks that have protected the field m thus far

through the execution. For example, a particular lock l is in the set $C(f)$ if every thread that has accessed field f was holding l at the the time of the access. When a new field m is initialized, $C(m)$ is set to the set of locks currently held by the thread which performs the initialization. At each access of m by a thread, the Eraser tool intersects $C(m)$ with the set of locks held by the accessing thread. The intersection operation refines the list to only contain the common locks held at every m access event. If $C(m) = \emptyset$ at the end of the program then the tool issues a warning.

The Eraser algorithm contains refinements so that it produces fewer false positives. Three safe programming idioms were discovered that produced false positives with the lock-set algorithm:

- *Initialization:* Shared fields are frequently initialized without a lock being held. This is safe because, typically, no other thread holds a reference to the object being initialized.
- *Read-only shared data:* State is initialized with a value and is read-only thereafter.
- *Read-write locks:* State is accessed by multiple readers, but only a single writer.

To support the first two programming idioms, Eraser uses a state machine, shown in Figure 2.3, to track actual use of a field. When a field is created, it is set to the *Virgin* state, indicating that the data is new and has not been referenced by a thread. Once the data is accessed by a thread it transitions to the *Exclusive* state. This means that at the present time only one thread has accessed the field. This addresses the initialization of $C(m)$, because the first thread can initialize the field without causing $C(m)$ to be refined. If another thread accesses the field, then the state changes. A read access changes the state to *Shared*. In the *Shared* state, $C(m)$ is updated, but race conditions are not reported. This addresses the read-only shared fields, because numerous threads can read a variable without writing to the field and not develop a race condition. The other case that needs to be addressed is when a thread writes to a field. A write access from a different thread changes the state

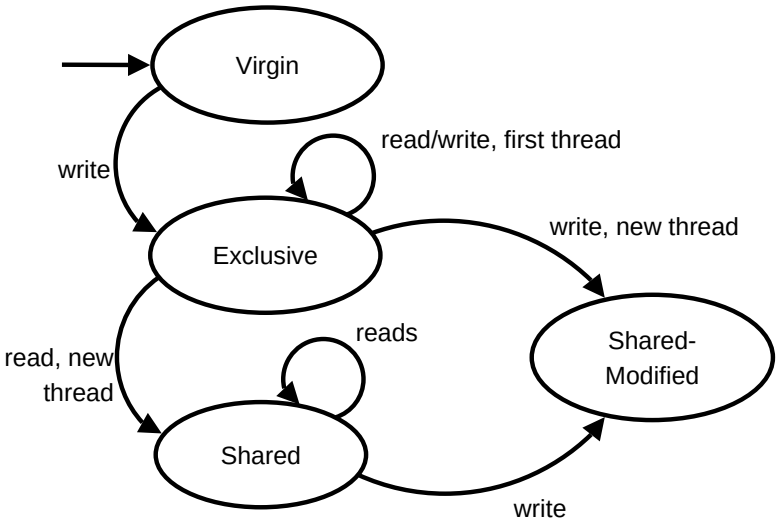


Figure 2.3: Eraser's state machine for memory locations [20]. Each new memory location starts in the *Virgin* state. Once a memory location is initialized with a value the state changes to *Exclusive* state. If another thread reads the value, the memory location transitions to the *Shared* state. As long as the memory location is just read it remains in the *Shared* state. If another thread writes to the memory location, the memory location transitions to the *Shared-Modified* state. In this state, potential race conditions are reported if all accesses to the memory location are not protected.

from *Exclusive* or *Shared* to *Shared-Modified*. In this state $C(m)$ is updated and race conditions are reported.

The third programming idiom uses locks with different modes to protect write and read accesses. As long as a thread holds one of the read locks, it is granted access to read the state. However, only threads holding a write lock are able to write to the state. The Eraser algorithm works by comparing which locks are held to perform reads and writes. To determine a potential race condition, locks held purely in read mode are removed from the candidate set of locks when a write occurs, because the locks used only to protect reads do not protect against race conditions between the writer and some other readers.

We make use of the classic lock-set algorithm used by Savage, et al in Eraser. We implement a modification of Eraser's state chart based on our quantum implementation. Our analysis incorporates the initialization and read-only modifications to reduce the number of false positives in typical Java code. These modifications allow

FlashLight to report more precise results about the behavior of the program when compared with the basic lock-set algorithm.

2.2.3 O'Callahan–Choi Hybrid. O'Callahan and Choi in [17] propose a hybrid dynamic race condition detection algorithm that combines happens-before and lock-set techniques. Their algorithm tries to reduce the false positives of the lock-set algorithm while at the same time keeping its overhead low. This work demonstrates the importance of tuning program instrumentation to reduce program execution time. They introduce a dynamic optimization "oversized-lockset" whereby they run the program twice, tuning instrumentation for the second run based upon results of the first run. Benchmark programs demonstrate these two runs combined are often far quicker than a single run without tuned instrumentation. For example, the Tomcat web server takes roughly 81 seconds to execute both runs when "oversized-lockset" is applied, but 129 seconds when a single run is made with full instrumentation. The empirical basis for the "oversized-lockset" dynamic optimization is that most Java threads hold very few locks at any point in time.

FlashLight does not implement the "oversized lockset" dynamic optimization proposed by O'Callahan and Choi, nor any form of "multi-run tuning" of dynamic instrumentation. Instead, our use of AspectJ to instrument the program allows direct programmer tuning of how much instrumentation is added to the program. We are unlikely to add "multi-run tuning" of FlashLight instrumentation in the style of O'Callahan and Choi because in our case studies we have encountered programs that are difficult to run in a repeatable manner. These programs include those with graphical user interfaces that must be manipulated by the programmer to ensure program progress, and application servers that require lengthly pre-execution set up.

This concludes our discussion of the dynamic race condition detection algorithms. The next two sections describe alternative race condition detection algorithms. The first technique uses abstraction to create a model of the program. The

second technique evaluates the structure of the code. In these sections, we explain how these two approaches differ from FlashLight.

2.3 Model Checking Techniques for Race Condition Detection

FlashLight suggests locking models that can be expressed and subsequently verified by the Fluid assurance tool. The Fluid assurance tool requires design intent that FlashLight tries to infer based upon the runtime behavior of the program. FlashLight also reports possible faults or "bugs" in the program (i.e., race conditions)—in this sense it is a "bug hunting" tool.

Tools based upon *model checking* are another approach to "bug hunting." These tools typically use static analysis to create abstract models of the code. These models are then run through a model checker, such as Spin [11], to locate potential concurrency faults. An example of a model checker tool is Java PathFinder2 [23], which is a custom-built model checker for Java. This tool was built in response to short-comings in previous model checkers that lacked the ability to model the entire language. It is a new model checker that is able to execute the entire language. JPF incorporates static analysis tools to reduce the state space that has to be searched by the model checker. The tool also has the ability to perform run-time analysis using two run-time algorithms, Eraser's lock-set algorithm and their own "LockTree" lock-set approach. These algorithms can be used stand-alone or with the model checker [23].

The concept of using runtime analysis to guide model checking is further discussed by Havelund in [10]. He describes an approach of integrating dynamic analysis with model checking to find race conditions and deadlocks. The tool has two operating modes. The first is a stand-alone or simulation mode that uses a dynamic analysis to report race conditions and deadlocks. The second mode generates reports about possible race conditions and deadlocks that can be used with their custom built model checker to evaluate consequences of the errors [10]. Much like FlashLight, both of these techniques use their run-time analysis to provide insight into the dynamic nature of a program.

2.4 Static Analysis Techniques for Race Condition Detection

There are numerous static analysis tools for locating shared state and race conditions. One static tool for detection of race conditions is RacerX [5]. This C-language tool is designed to locate errors in large, complex multi-threaded systems (e.g., operating systems, which are typically implemented in C). It uses a flow-sensitive, interprocedural analysis to locate both deadlocks and race conditions. This tool operates on code with no additional design intent to "hunt bugs." It is both unsound and incomplete. RacerX has, however, uncovered faults in several operating systems.

A hybrid static–dynamic technique for race condition detection proposed by von Praun and Gross in [18] is based on object race detection instead of field accesses. Their detector is designed to locate races in object access opposed to field access. An object access occurs when a method of an object is called. The detector uses the concept of confinement as described by Lea in [15]. Confinement is a property of a program that exploits encapsulation of data to guarantee that at most one thread can access an object. Confinement is used to reduce the amount of program instrumentation because the structure of the object accesses can be determined at compile-time. They make use of static analysis techniques, namely escape analysis, to determine which objects could be shared. The dynamic analysis determines which objects are accessed by multiple threads and if any of these accesses lead to potential race conditions.

von Praun and Gross use an object use graph (OUG) to statically capture accesses from different threads to objects for the purpose of detecting race conditions [19]. The OUG approximates Lamport's happens-before relation between access events issued by different threads to a specific object. This technique locates object races as opposed to field races as in many other techniques, including our own. The information in the OUG has been used to instrument Java programs with dynamic checks for object races.

Determining whether two field accesses could happen simultaneously is an important step in identifying a possible race condition. The *may happen in parallel* relationship is applicable to optimization, anomaly detection (e.g. race conditions), and improving accuracy of data flow analysis. Naumovich, Avrunin, and Clarke in [16] describe a data flow method for computing a conservative approximation of the set of pairs of statements that may happen in parallel in a Java program. Their algorithm has a worst case bound that is cubic in the number of statements in the program.

2.5 Engineering Dynamic Analysis using AOP

FlashLight uses aspect-oriented programming (AOP) to instrument code to gather run-time information about field accesses and lock acquisition. Section 2.5.1 provides an overview of AOP. Section 2.5.2 discusses some other dynamic analysis approaches that use AOP.

2.5.1 An Overview of AOP. Kiczales, et al. provides the foundation for aspect-oriented programming in [13] and background on the development of the AspectJ language, which we use for FlashLight, in [12]. The key problem AOP is designed to solve is how to handle cross-cutting concerns within an application. The cross-cutting concerns are the result of composing an application in two different manners because of restrictions placed on the developer by the programming language.

The central element of any aspect-oriented language is the join point model. Join points are well-defined points in the execution of a program. Join points can be considered as nodes in a simple runtime object call graph. These nodes consist of points at which objects receive calls, objects are constructed, and objects are referenced. The edges of the call graph are control flow relations between the nodes. In this graph, control passes through each node twice, once on the way in and once on the way out—that is, before and after the join point.

A *pointcut* specifies a set of join points. AspectJ provides primitive pointcuts to be used to match the join points. Pointcuts can also be composed to match more

complex join point expressions. *Advice* is a segment of code associated with a pointcut that is executed when a join point is matched. Advice can be inserted into three positions for each join point, *before* a join point, *after* a join point, or both, called *around* advice. Pointcuts are combined with advice to form *aspects*. Aspects are defined similarly to classes. Aspect declarations may include pointcut declarations, advice declarations, and any other declaration allowed in class declaration.

To make advice easier to construct, AspectJ provides a reflexive capability to the current join point. Within advice, the special variable `thisJoinPoint` is linked to the object representing the current join point. This object provides information common to all join points (e.g., kind and signature of the join point). The `thisJoinPoint` also provides information specific to each kind of join point: for example, a field access join point provides information about the field signature.

A goal of any AOP language is to have the aspect and regular code execute in unison. This coordination process is called *aspect weaving* and involves insuring that advice executes at the appropriate join points. AspectJ provides a compiler-based implementation to perform the weaving. This implementation performs almost all weaving work at compile-time. There are a few advantages to this compile-time implementation. First, it exposes as many errors as possible at compile time. By integrating the tool into an IDE, this provides prompt user feedback. Second, this implementation avoids unnecessary runtime overhead (i.e., checking at all points in the call graph if advice needs to be run).

The AspectJ compiler uses a "pay-as-you-go" strategy. Code that is not affected by advice is compiled just as it would be by a standard Java compiler. The AspectJ compiler transforms advice into a standard Java method that is run before or after the join point (as specified by the pointcut for its corresponding aspect).

2.5.2 Other uses of AOP for Dynamic Analysis. Our use of AspectJ in particular, and AOP in general, as the vehicle to instrument a program is not novel.

However, it is not yet common practice. In this section, we review some prior dynamic analysis work which, like our work, relies upon AOP to instrument a program.

Bierhoff and Aldrich in [1] use AspectJ to ensure objects at runtime conform to a specified protocol, which they term a *typestate*. Their tool uses AspectJ to instrument existing Java code with dynamic checks of conformance to the programmer's typestate specification.

Goldberg and Havelund describe their custom built instrumentation package JSpy in [6]. JSpy is designed to instrument code to locate race conditions and deadlocks. JSpy was developed because AspectJ is unable to determine the boundaries of synchronized statements. Our solution, discussed in more detail in Chapter IV, is to rewrite the source code around synchronized statements in the program to be analyzed.

Boroday, et al. designed a dynamic anti-pattern detector which they describe in [2]. Their work uses AOP for program instrumentation. They convert the output from an instrumented program into a Promela model and use the Spin model checker to verify the code is free of anti-patterns including race conditions. Similar to FlashLight, the dynamic analysis portion of this tool is intended to feed into a verification system—in their case to the Spin model checker, in our case to the Fluid assurance tool. A key difference is that Boroday, et al. define the anti-patterns (i.e., design intent) that Spin searches for violations of. FlashLight guesses design intent by proposing a lock model for each piece of consistently protected state in the program. However, we require a "programmer in the loop" who can refine or reject the model proposed by FlashLight before asking the Fluid assurance tool to perform a verification of model–code consistency. Thus, we as tool developers do not, *a priori*, try to impose design intent upon a concurrent system (i.e., what constitutes an anti-pattern).

III. Tool Use

This chapter describes how a programmer would use FlashLight to better understand the concurrency in their program. FlashLight use can be divided into three steps:

1. Customize FlashLight instrumentation.
2. Run the target program with FlashLight instrumentation.
3. Examine reports about the target program's concurrency.

We describe each of these steps in this chapter. Section 3.1 describes how to customize FlashLight's instrumentation. Section 3.2 describes running the instrumented program. Finally, Section 3.3 describes the set of reports produced by FlashLight about the target program.

3.1 Customizing FlashLight Instrumentation

FlashLight requires information about how to instrument a target program. Specifically, the programmer needs to tell FlashLight when the analysis should start and stop collecting data. FlashLight allows multiple time periods of dynamic data collection, called quantums. These are partitions of the running program's timeline. Quantums allow the programmer to analyze parts of the program's execution separately, e.g., this is the "start up" phase of my program, this is the "steady state" of my program, and this is the "shut down" phase of my program. To lower runtime overhead, the programmer may also restrict data collection to a subset of the program's classes. The programmer provides information about how to instrument a target program in the form of AspectJ pointcut specifications. FlashLight then uses the AspectJ compiler to "weave" these instrumentation specifications into the target program.

To track lock acquisitions within the program, a source code rewriter that inserts additional instrumentation is run on the program. This source code rewriter is needed because, as is discussed further in Chapter IV, the pointcut mechanism of AspectJ

cannot track lock acquisition and releases within a program. The process for initiating the source code rewriter is described in detail in Section 3.1.1.

3.1.1 Setting Up FlashLight. During our development and case studies we used FlashLight within the Eclipse IDE with the AspectJ Development Tools (AJDT) plug-in. FlashLight can, however, be run outside of Eclipse. This capability was used in our commercial case study described in Section 5.3 on page 76. FlashLight requires the AspectJ compiler to weave advice into the target program and generate instrumented byte code. FlashLight also requires a Java Runtime Environment (JRE) to execute the instrumented program. The following directions assume the programmer is using the Eclipse IDE. Our own experience confirms that FlashLight is portable to both the Linux and Windows operating systems.

1. Install a Java SDK (available at `http://java.sun.com`), the Eclipse Java IDE (available at `http://www.eclipse.org`), and the AspectJ AJDT (available at `http://www.eclipse.org/aspectj`).
2. Install the FlashLight source code rewriter in the Eclipse plug-in directory. The rewriter code can be checked out from from the CVS pserver host `fluid.cs.cmu.edu` from the repository path `/cvs/afit` using the module name `edu.afit.fluid.dynamic.rewriter`. This adds a menu choice, "AFIT Dynamic Lock Tracking," to every Java project that rewrites the project's source code to track lock acquisition and release.
3. Load the target code into an Eclipse project. Ensure that you *make a copy* of the original code. This is important because the FlashLight source code rewriter changes the original code and our current implementation does not allow the changes to be reversed (this is a straightforward feature to implement but was not done due to time constraints).
4. Check out the FlashLight code, as an Eclipse project, from the same CVS server used to install the rewriter. This code is stored under the module name

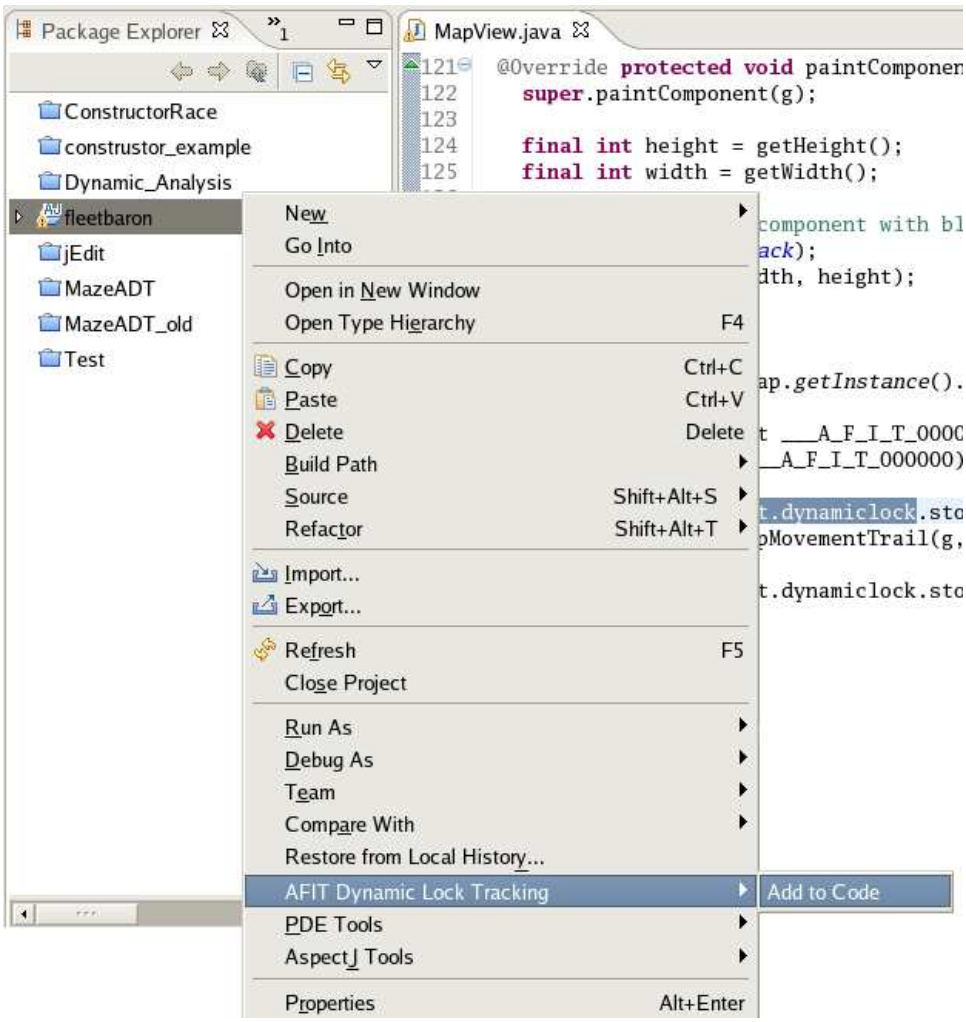


Figure 3.1: Invoking the FlashLight Source Code Rewriter. This menu action rewrites the source code of the `fleetbaron` project to allow FlashLight to track lock acquisitions and releases by threads within the running target program.

`/shale/Dynamic_Analysis`. This project represents the parts of the FlashLight code that must be added to the target code to preform FlashLight 's dynamic analysis.

5. Copy the source folder "Analysis_Tools" from the "Dynamic_Analysis" project into the project containing the target code.

6. Run the FlashLight source code rewriter on the target code's project by selecting "AFIT Dynamic Lock Tracking" → "Add to Code" as shown in Figure 3.1.

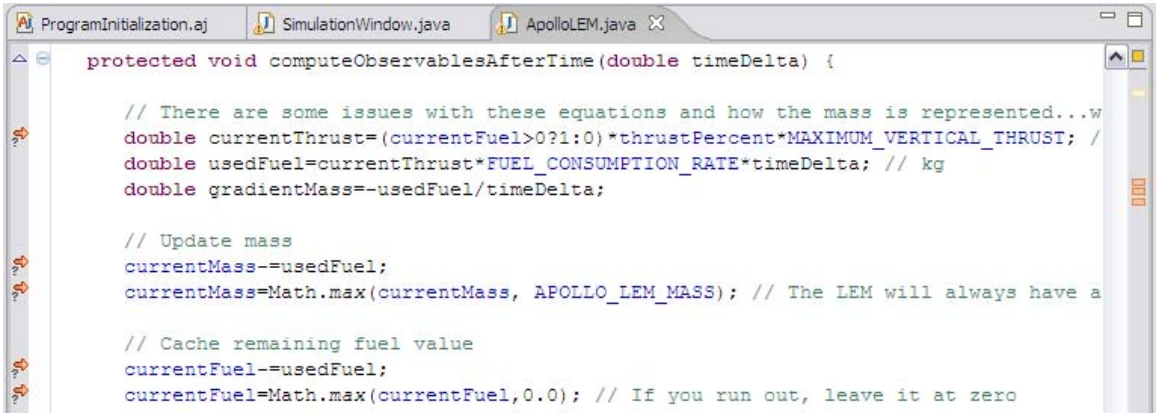


Figure 3.2: AspectJ-Specific Icons. The appearance of red arrow icons (to the left) within the target code is a good indication that the target program is instrumented and ready to be run. If they don't appear, a rebuild of all the code with AspectJ may be required; alternatively, the instrumentation specification may be inconsistent with the target program's source code.

7. Add the "Aspect Nature" to the project containing the target code by right-clicking on the project and selecting "AspectJ Tools" → "Add AspectJ Nature" (like the previous step). This step allows the project containing the target code to be compiled using the AspectJ compiler that FlashLight uses to "weave" its instrumentation into the target program.

8. When the target program is run, FlashLight will place its output reports into a folder named `xml`. The `xml` folder contains the files to transform and present the XML output generated by FlashLight as programmer readable web page reports. To setup this folder, you unzip the `xml.zip` file located at the root of the "Dynamic_Analysis" project into your project.

9. As introduced above, FlashLight needs to be provided with a program-specific instrumentation specification. We cover this topic in further detail below.

10. At this point, there should be no errors in the project. If Eclipse does not update itself with AspectJ-specific icons, as shown in Figure 3.2, rebuild the workspace.

11. Run your application and exercise it as you wish. During the program's execution FlashLight will collect data per the instrumentation specification.

12. Upon the successful termination of your program you may need to refresh the Eclipse "Package Explorer" view. This makes the FlashLight reports appear in the `xml` folder. You can examine the reports about the execution of the target program by opening the `index.html` file in any web browser.

Once the AJDT and the rewriter plug-in are installed into Eclipse, only steps 3 through 12 are required to configure FlashLight to analyze a different target program.

3.1.2 Tuning Target Program Instrumentation. Tuning the target program instrumentation consists of introducing several AspectJ pointcut specifications to control aspects of FlashLight's instrumentation. The program initialization aspects "turn on" FlashLight, meaning they create quantums and allow the FlashLight data store to capture data from the instrumentation. The program termination aspects stop data capture and cause FlashLight to analyze its collected data and output the reports about the target program. These aspects are specialized for each target program and require programmer insight about the runtime behavior of the target program to obtain useful results from FlashLight.

This section describes several helpful patterns for tuning FlashLight target program instrumentation. These patterns emerged during our case studies. First, we describe pointcuts used to start FlashLight data collection. Second, we explain how to advance the quantum (optionally without data collection). Finally, we discuss effective ways to terminate data collection, execute the analysis of the collected data, and output FlashLight reports.

- **Useful pointcut patterns:** We must define pointcuts to weave in advice to advance the collection quantums. One typical situation is to start data collection when a class is initialized. We developed a pattern of using the *staticinitialization* pointcut, which matches any class that is initializing. For example, the declaration

```
pointcut startup() : staticinitialization(*..Maze);
```

indicates we want to "trigger" when the `Maze` class is initialized (i.e., loaded by the class loader). This pattern is useful if you want to start FlashLight data collection right at the very start of a program. To do this replace `Maze` with the name of the class containing your main program.

Another pattern encountered is that the programmer wants to delay data collection until the target program completes initialization and transitions into a "steady state." We have found this pattern useful for network servers and programs with significant graphical user interfaces because these types of programs have a clear "start up" phase (which is single threaded) followed by a concurrent "steady state." We specify a pointcut that executes after the program is fully initialized. For example, the declaration

```
pointcut steadyState() : call(* *..*.setVisible(..));
```

indicates we want to "trigger" when the `setVisible` method is invoked. In a program using the Swing framework, this call is typically used to make the main window of the application visible on the screen.

- **Advancing the quantum:** Using the pointcuts we just discussed, we can now describe how we advance quantums. The pointcut is the trigger and the calls discussed in this section control FlashLight data collection. Quantums partition the program execution. Quantums act as a container for all target program data FlashLight collects, and reports are generated for each quantum that contains data. The instrumentation triggers when quantums begin by simply advancing the quantum. The new quantum is in effect until the instrumentation advances to a new quantum, or collection is terminated. There are two methods that advance a quantum. The first, *advanceQuantumNoCollection*, advances the quantum but does not collect data for the new quantum. The second, *advanceQuantumWithCollection*, advances the quantum and does collects data for the new quantum. For example, the declaration

  ```
  pointcut startUp() : staticinitialization(*..Main);
  ```

```
before() : startUp()  {
  Store.getInstance().advanceQuantumNoCollection();
}
```

advances the quantum with no collection when the `Main` class of the target program is initialized. We use this approach to start FlashLight and skip data collection until the program reaches its "steady state" phase of execution. At that time we advance the quantum and begin to collect data. For example, the declaration

```
pointcut steadyState() : call(* *..*.setVisible(..));

before() : steadyState()  {
  Store.getInstance().advanceQuantumWithCollection("SteadyState");
}
```

starts a new quantum, called `SteadyState`, with data collection when the `setVisible` method is invoked.

The *advanceQuantumWithCollection* method takes two parameters. The first is mandatory but the second is optional. The first parameter provides a programmer-defined name for the quantum (the example above defines `SteadyState` as the quantum name). The second parameter allows the programmer to specify a prefix for all report filenames (the example above doesn't define a report filename prefix). This optional prefix is useful for target programs that have multiple main programs. It provides a way to distinguish each main program's FlashLight reports.

- **Generating output reports:** A programmer specification of when FlashLight should stop data collection, analyze its data, and output reports is *mandatory*. If the program terminates before this aspect is triggered, then all collected data is lost. Consider the declaration

```
pointcut shutdown() : call(* *..System.exit(..));}

before() : shutdown() {
  Store.getInstance().systemOutput();
}
```

that stops FlashLight before any call to `System.exit()` that occurs in the program. This approach works well with most graphical applications.

Another typical pattern, which is useful for non-graphical Java programs, is to stop FlashLight after the `main` method of the program finishes its execution.

```
pointcut shutdown() : execution(* Main.main(..));

after() : shutdown() {
  Store.getInstance().systemOutput();
}
```

In both cases, the termination aspect calls the *systemOutput* method to direct FlashLight to finish up and output its reports.

3.2 Running the Target Program

The programmer can invoke a large test suite or put the instrumented program into any "production-like" situation he or she deems of interest. The goal is to stimulate the execution of as many of dynamic paths within the program as possible so that FlashLight can produce the best possible results for the programmer. FlashLight collects data as the program runs and creates web page reports about that particular program execution.

3.3 Examining FlashLight Reports

FlashLight produces a suite of web page reports that a programmer can examine to better understand the target program's concurrency. Each instrumented program generates four XML data files reporting the results of the analysis. XSL files are used to present the XML file data in a web browser to the programmer. The web page presentation of FlashLight results is currently the only method of viewing tool output. However, we selected XML as the format of the tool's output to facilitate other views of the tool results in the future (e.g., a view of FlashLight results within the Fluid assurance tool).

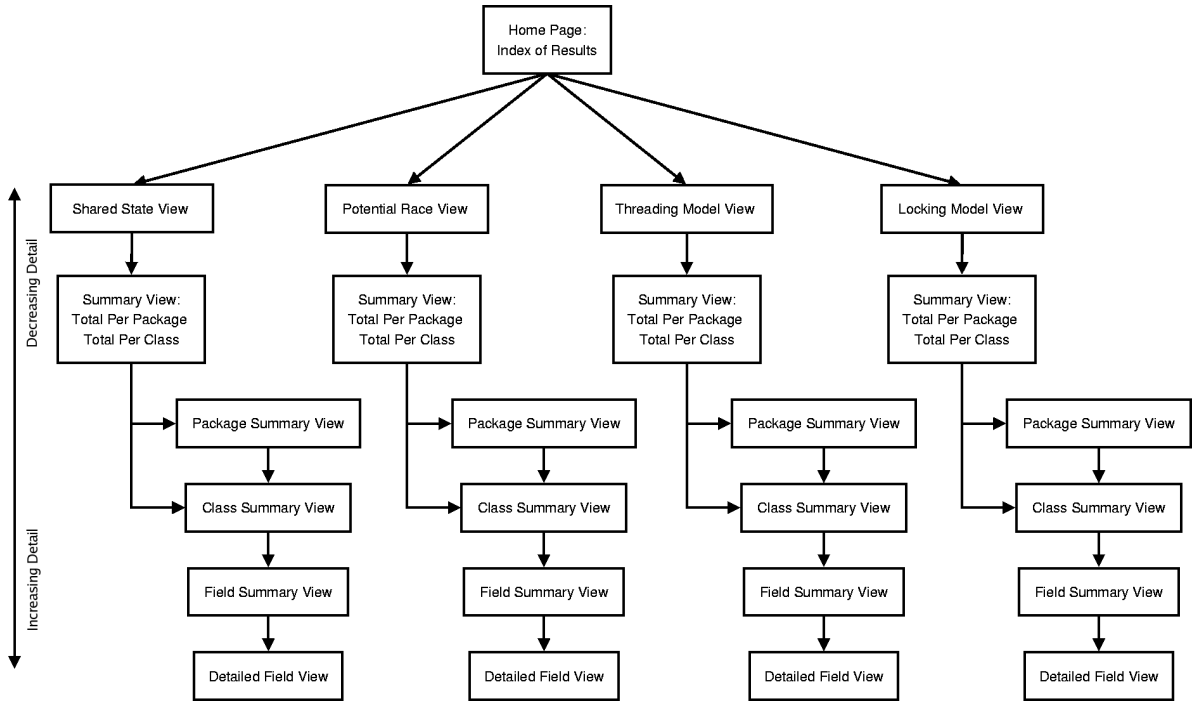


Figure 3.3: Structure of FlashLight Reports.

Analysis Output

- UnProtected MazeADT -- Steady State Tracking Quantum -- SharedState
  Reports all fields accessed by multiple threads
- UnProtected MazeADT -- Steady State Tracking Quantum -- PotentialRaceDetection
  A subset of the Shared State View--Reports all fields accessed by multiple threads in an inconsistent way
- UnProtected MazeADT -- Steady State Tracking Quantum -- ThreadingModel
  A subset of the Shared State View--Reports all fields that are accessed by threads holding a common set of locks and suggests possible locking annotations
- UnProtected MazeADT -- Steady State Tracking Quantum -- LockingModel
  Reports which objects lock access to shared fields and suggests possible locking annotations

Figure 3.4: Results Home Page. This screen shot shows the home navigation page for the results. This file list each output file associated with this execution of FlashLight.

• **Field requestCount in class org.gjt.sp.util.WorkThreadPool**

Instance	Thread Name	Read Count	Write Count
WorkThreadPool			
	AWT-EventQueue-0	29	2
	jEdit I/O #2	2	1
	jEdit I/O #4	2	1

Figure 3.5: Shared State. This report lists all field accesses by multiple threads where at least one thread writes to the field. This screen shot shows field `requestCount` and the three threads that accessed the field.

Figure 3.3 shows how FlashLight results are organized into four separate views. The top level of each view summarizes the fields by package and class and the programmer can "drill down" to obtain more detail about a result of interest. From any level the user can return to the top of the current page or the home page which is shown in Figure 3.4. We now describe the contents of each report "view."

- **Shared state:** This report lists all the fields that are accessed by multiple threads regardless of locking protection. It reports any field that is accessed by at least two threads where at least one access writes a value to the field. The example in Figure 3.5 shows the field `requestCount` within the only instance of the `WorkThreadPool` class has been accessed by three threads. The report uses links to navigate through regions of the page. The underlined `WorkThreadPool` object instance shown in Figure 3.5 is a link taking a programmer to more detailed information about the field (within that instance), including stack traces to help the programmer understand precisely how the state was shared and by which threads.
- **Potential races:** This report lists all the fields that are accessed by multiple threads where, at the time of access, no common lock is held by all the threads. In addition to the inconsistent locks held, this view requires a field to be shared. In Figure 3.6 we see the same field from Figure 3.5, `requestCount`, only this report has categorized the field as a potential race condition based

• **Field requestCount in class org.gjt.sp.util.WorkThreadPool**

Instance	Thread Name	Read Count	Writes Count	Locks Held by Thread
WorkThreadPool				
	AWT-EventQueue-0	29	2	■ No lock is held at this field access
	jEdit I/O #2	2	1	■ Lock <lock>. java.lang.Object@dd3b71
	jEdit I/O #4	2	1	■ Lock <lock>. java.lang.Object@dd3b71

Figure 3.6: A Potential Race Condition. This report lists all fields that are not consistently protected by locks. This screen shot shows the field `requestCount` has been accessed by three threads. The threads `jEdit I/O #2` and `jEdit I/O #4` held the lock *lock* but the `AWT-EventQueue-0` thread did not hold a lock.

• **Field m_isMoving in class edu.afit.fleetbaron.common.game.Ship**

Locks consistently held by threads accessing field: **m_isMoving**

○ @lock m_isMovingLOCK is <this>.Ship@1de6817 protects m_isMoving

Instance	Thread Name	Read Count	Write Count
edu.afit.fleetbaron.common.game.Ship@1de6817			
	client handler p1	7	6
	TurnCyclicBarrier	327	5

Figure 3.7: A Proposed Lock Model. This report lists all fields that are consistently protected by locks. This screen shot shows the field `m_isMoving` has been accessed by the threads `client handler p1` and `TurnCyclicBarrier`. Both threads held a lock on the `Ship` instance (which contains the field) when they accessed the field. FlashLight has proposed a possible lock policy for this field via the Greenhouse-style lock policy annotation *@lock*.

on inconsistent locking by threads during accesses. We see the threads `jEdit I/O #2` and `jEdit I/O #4` held the lock *lock* when accessing the field but the `AWT-EventQueue-0` thread did not hold a lock during any of its accesses.

- **Threading model and Locking model:** This report contains two different views of the same data. The threading model view reports consistently protected fields based on what locks were held by the threads which accessed the fields. The locking model view reports which locks consistently protected each field. We see in Figure 3.7 the field `m_isMoving` was protected by holding a lock on

its enclosing `Ship` instance. Both threads perform multiple reads and writes. FlashLight cannot know the design intent of the developer with regard to how accesses to the shared `m_isMoving` field should be protected. However, based upon what it has observed, FlashLight suggests a possible lock policy model using the *@lock* annotation. This annotation should be viewed as a starting point for program verification using the Fluid assurance tool.

For cases when FlashLight determines that a shared field is consistently protected, FlashLight suggests a locking policy for that field. This proposed locking policy may or may not align with programmer intent (assuming such intent exists or is remembered). FlashLight proposes a locking policy via a "dynamic" *@lock* annotation. There is an "impedance mismatch" between the dynamic view of the lock policy and the static view of the lock policy that the programmer must reconcile, especially with respect to the proposed lock object. The two types of "dynamic" *@lock* annotations reported by FlashLight are

1. @singleThreaded – this lock will be reported when a field is written during object creation (i.e., field declaration, constructor, initializer block, etc.) and all other access are read accesses.

2. @lock – used when all threads accessing a field hold a common lock. Unlike the exact Fluid annotation that allows a lock to protect an abstract grouping of fields, this notation declares an object protects a single field. For example,

   ```
   @lock firstReqLOCK is <lock>.java.lang.Object@10c99
     protects firstRequest
   ```

   means FlashLight has noted that a lock on the object `lock` is consistently held by threads when they accessed the field `firstRequest`. Similar to the "static" *@lock* notation we give the proposed "dynamic" lock an explicit name, `firstReqLOCK` in this example. There are two parts in our "dynamic" lock policy notation to identify the lock: the context and the referenced object. We refer to the first part as the *context*—how the ob-

ject is used to protect access. The context appears within the <>. There are three types of contexts used in FlashLight `this`, `CLASSNAME.class`, or `OBJECTNAME`.For a field protected by the current instance object (e.g., by synchronized methods), the context reported is `this`. For a field protected by locking on a class instance, the context of the lock is `CLASSNAME.class`, where CLASSNAME is replaced by the actual name of the class. When a field is protected by an object other than the current object, we use the name of the object reference as the context. In the above example, an object is protecting access to `firstRequest`, therefore the context is the name of the reference, `lock`. The second part of the lock identifies the (dynamic) *referenced object*. In the above example, the referenced object is of type `Object` and has id `10c99` in the running program's heap.

There are times when FlashLight finds that more than one lock protects a field. In cases where multiple locks protect a field, FlashLight does not guess which one is actually intended by the programmer. Instead, all of the locks consistently held during field accesses are reported for programmer consideration. In the output

```
@lock yCoordLOCK is <this>.Ship@1de6817  protects yCoord
@lock yCoordLOCK is <@singleThreaded>    protects yCoord
@lock yCoordLOCK is <this>.Thread.135324 protects yCoord
```

the *this* context is ambiguous. It is for this reason we append the reference object onto the context.

Finally, we caution that FlashLight infers lock policy models *based on only one execution of a program*. Thus these proposed models are intended to be a starting point, not a final model, for performing program verification using the Fluid assurance tool.

3.4 Summary

This chapter presents, in three parts, how to use the FlashLight tool. First, a user sets up the tool and tunes program specific instrumentation. While these

aspects are unique to each application, we present patterns which we have found helpful when working with several target programs. Second, a user runs the target program. Third, the user examines reports about the target program's shared state, potential race conditions, and proposed locking models. The proposed models can be used as a starting point to assure aspects of the target program's concurrency design intent using the Fluid assurance tool.

IV. Tool Engineering

FlashLight is composed of three components that collaborate to collect the data, store the data, analyze the collected data, and output the results of the analysis in the form of programmer reports. These components, shown in Figure 4.1, are

1. The *instrumentation* that monitors the running program triggering necessary data collection.
2. The *data store* that holds and organizes the collected data.
3. The *analysis* that examines the collected data and creates output reports for the programmer.

The next three sections of this chapter describe each of these components in turn.

4.1 The Instrumentation

FlashLight's instrumentation monitors the running program triggering necessary data collection. In this section we describe the design and implementation of the tool's instrumentation. FlashLight uses two technical approaches to instrument the running target program:

1. AspectJ, which we use to instrument field reads and writes, as well as to instrument special lock acquisition and release method calls.

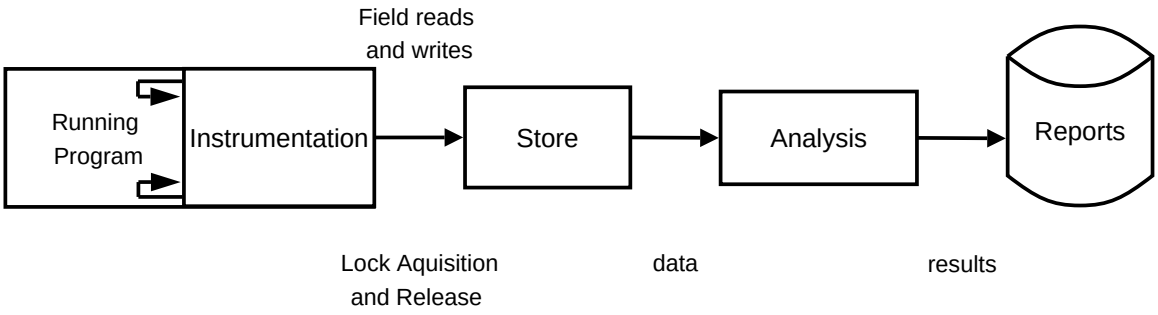


Figure 4.1: An Overview of FlashLight's Components.

```
1  pointcut readObject()  : get(Object+*) &&
2                           within(!edu.afit.dynamiclock.store..*);
3  pointcut writeObject() : set(Object+*) &&
4                           within(!edu.afit.dynamiclock.store..*);
5
6  pointcut readPrimitive()  : (get(int *)  || get(double *)|| get(float *) ||
7                               get(byte *) || get(short *) || get(long *)  ||
8                               get(char *) || get(boolean *)) &&
9                               within(!edu.afit.dynamiclock.store..*);
10 pointcut writePrimitive() : (set(int *)  || set(double *)|| set(float *) ||
11                              set(byte *) || set(short *) || set(long *)  ||
12                              set(char *) || set(boolean *)) &&
13                              within(!edu.afit.dynamiclock.store..*);
```

Figure 4.2: Pointcuts Matching Field Reads and Writes. Lines 1–2 match all reads of reference fields and lines 3–4 match all writes to reference fields. Lines 6–9 match all reads of primitive type fields and lines 10–13 match all writes to primitive type fields. To instrument the target program only, and not FlashLight's code, each pointcut definition specifies that a match should not occur if the field access is within the packages that contain the FlashLight source code.

```
after() : readObject() {
  if (Store.getInstance().collecting()) {
    JoinPoint tjp = thisJoinPoint;
    Store.getInstance().addFieldRead(tjp.getSignature().getDeclaringType(),
                                     tjp.getTarget(),
                                     tjp.getSignature().getName(),
                                     Thread.currentThread());
  }
}
```

Figure 4.3: Advice for a Field Read. When AspectJ detects a read of reference variable, it calls the addFieldRead method to direct the FlashLight data store to record the data. This method receives the class of the object, the object containing the field, the field name, and the thread that performs the read.

2. Source code rewriting, which we use to convert synchronized blocks into pairs of method calls that signal lock acquisition and release.

AspectJ is our primary source of instrumentation. We use source code rewriting to overcome a deficiency in the expressiveness of AspectJ 's pointcuts. In the following subsections, we describe how we use aspects to collect information about field accesses, how we use a combination of source code rewriting and aspects to track the set of locks each thread holds, and how we support the common Java programming idiom of not locking during object initialization.

4.1.1 Detecting Field Reads and Writes. FlashLight uses AspectJ to capture every field read and write. Our instrumentation captures every field read or write made by the running program. AspectJ provides the pointcut `get` to match the join points for all field reads and the pointcut `set` to match the join points for all field writes. FlashLight uses four pointcuts to capture all of a program's field access; these are shown in Figure 4.2.

The advice (i.e., the code triggered by a field read or write) reports data to the FlashLight store as shown in Figure 4.3. The data is only reported if the store is currently collecting data. The data store is collecting data when its `collecting` method returns *true*.

It is possible to tune the field instrumentation to record data for specific classes or packages only within a target program. The programmer would do this by adding more `within` restrictions to the pointcuts shown in Figure 4.2. These restrictions would be syntactically similar to the pointcuts that currently exclude the FlashLight source code. We used this type of tuning during our commercial case study to exclude several utility packages that were uninteresting from the point of view of concurrency.

4.1.2 Tracking Locks. Instrumentation to track the set of locks each thread holds is done using both AspectJ and source code rewriting. Source code rewriting is required because an AspectJ pointcut can not "trigger" advice at the beginning and end of a `synchronized` method or block. This is a known limitation of the AspectJ language. To solve this problem, we constructed a source code rewriter for FlashLight that introduces identifiable method calls that our AspectJ instrumentation is able to trigger on.

An example of the transformations the source code rewriter performs is shown in Figure 4.4. The rewriter is implemented in a manner similar to an Eclipse refactoring and is invoked as shown in Figure 3.1 (on page 36). The rewriter uses a flow-insensitive intra-procedural static analysis to find every instance of the `synchronized` keyword and transforms its associated method or block. The transformation inserts method

```
public class RewriterDemo {
  final Object lock = new Object();
  synchronized void m1(){
    // do something
  }
  static synchronized void m2(){
    // do something
  }
  void m3() {
    synchronized(lock){
      //do something
    }
  }
}
```

```
public class RewriterDemo {
  final Object lock = new Object();
  synchronized void m1(){
    try {
      edu.afit.dynamiclock.store.LocksHeld.acquire(this, "this");
      // do something
    } finally {
      edu.afit.dynamiclock.store.LocksHeld.release();
    }
  }
  static synchronized void m2(){
    try {
      edu.afit.dynamiclock.store.LocksHeld.acquire(Demo_Rewriter.class,
        "Demo_Rewriter.class");
      // do something
    } finally {
      edu.afit.dynamiclock.store.LocksHeld.release();
    }
  }
  void m3() {
    {
      java.lang.Object ___A_F_I_T_000000 = lock;
      synchronized(___A_F_I_T_000000){
        try {
          edu.afit.dynamiclock.store.LocksHeld.acquire(___A_F_I_T_000000, "lock");
          //do something
        } finally {
          edu.afit.dynamiclock.store.LocksHeld.release();
        }
      }
    }
  }
}
```

Figure 4.4: Rewriting the `RewriterDemo` Class. The original class is shown above its output from the FlashLight source code rewriter. The `RewriterDemo` class contains code that triggers each of the three transformations performed by the FlashLight source code rewriter: (1) a synchronized method, (2) a static synchronized method, and (3) a synchronized block. The inserted FlashLight calls denote the boundaries of when a lock is acquired and released. The `try-finally` blocks are introduced to ensure that variable names are not masked and that the program's exceptional behavior is unchanged.

```
pointcut lockAcquire() : call(* edu.afit.dynamiclock.store.LocksHeld.acquire(..));
pointcut lockRelease() : call(* edu.afit.dynamiclock.store.LocksHeld.release(..));

after() : lockAcquire() {
  JoinPoint tjp     = thisJoinPoint;
  String filename   = tjp.getSourceLocation().getFileName();
  String linenumber = String.valueOf(tjp.getSourceLocation().getLine());
  Object[] callArgs = thisJoinPoint.getArgs();
  LocksHeld.acquireLock(callArgs[0], (String) callArgs[1], filename, linenumber);
}
after() : lockRelease() {
  LocksHeld.releaseLock();
}
```

Figure 4.5: Pointcuts and Advice for Lock Acquisition and Release. The `lockAcquire` advice captures the object being locked, the context of how the object is being used, and the filename and line number of the lock acquisition. The `lockRelease` advice "pops" the lock from our set of locks held by the thread which released it.

calls into the source code providing AspectJ access to the object being locked and the name or context of the locking object (as discussed below). The context of the locking object is used to provide insight into how the locking object is being used to protect the field. The position of the inserted calls frames the duration during which the lock is held.

With the rewritten source, we can now use AspectJ to collect when locks are acquired and released by each thread within the running program. AspectJ uses a `call` pointcut to match join points associated with the lock acquisition and release calls inserted by the FlashLight source code rewriter. The data store maintains a list of locks held for each thread. Figure 4.5 shows the lock acquisition and release pointcut and advice. You may wonder why we use a combination of source code rewriting and AspectJ to handle synchronization when it would appear that source code rewriting could be used exclusively. We still make use of AspectJ in this case because we can make use of *dynamic* information within advice that would not be available to the static source code rewriter.

4.1.3 Tracking Object and Class Initialization. In Java, it is typical that programmers do not protect object (and class) initialization by locking. This apparent

```
pointcut initGet()       : cflow(initialization(*.new(..))) && get(Object+*)
pointcut staticInitGet() : cflow(staticinitialization(*))   && get(Object+*)

pointcut initSet()       : cflow(initialization(*.new(..))) && set(Object+*)
pointcut staticInitSet() : cflow(staticinitialization(*))   && set(Object+*)
```

Figure 4.6: Initialization Pointcuts. These pointcuts match all field reads (`get`) and writes (`set`) that occur during object or class initialization.

violation of locking discipline is, however, safe in most cases. The practice is safe during construction because only the thread that invoked the constructor has access to the object's state, i.e., the object doesn't become shared state until after it is fully constructed. This practice becomes unsafe only if the constructor, while it is running, leaks a reference to the object under construction to another thread.[1]

To accommodate this idiom, we define additional advice that executes before and after our normal field access advice. We add a fake @singleThreaded lock to the set of locks held by the current thread. This fake lock communicates to the FlashLight analysis that the field read or write occurred within the boundaries of a Java constructor or initialization block.

Figure 4.6 shows the pointcuts we use to detect field reads and writes during class or object initialization. The instrumentation uses two additional AspectJ pointcuts `staticinitialization` and `initialization`. The `staticinitialization` pointcut captures class creation while `initialization` pointcut captures object creation. Aspect advice can be executed *before* or *after* a join point. The aspects in Figure 4.7 take advantage of this capability to acquire and release the @singleThreaded lock.

4.2 The Data Store

The FlashLight data store, or more simply "the store", organizes and stores the collected data in a manner that facilitates its subsequent analysis. The store is implemented in Java, not AspectJ. We made a design decision to limit AspectJ code

[1]While artificial Java programs that leak references to objects under construction are straightforward to construct, the Fluid team has only noticed this in real code when an object under construction registers itself as an observer to some (concurrent) component.

```
before() : initSet() || staticInitSet() {
  JoinPoint tjp = thisJoinPoint;
  ...
  if (tjp.getThis() == null){ // class initialization
    LocksHeld.acquireLock(tjp.getSignature().getDeclaringType(),
                          Store.getInstance().getLockingString());
  } else { // object initialization
    LocksHeld.acquireLock(tjp.getThis(), Store.getInstance().getLockingString());
  }
  Store.getInstance().addFieldWrite(tjp.getSignature().getDeclaringType(),
                                    tjp.getTarget(),
                                    tjp.getSignature().getName(),
                                    Thread.currentThread(), true);
}

after() : initSet() || staticInitSet() {
  LocksHeld.releaseLock();
}
```

Figure 4.7: Initialization Field Write Advice. This advice triggers *before* and *after* the join points matched by the pointcuts shown in Figure 4.6. The `before()` advice acquires the @singleThreaded lock (represented by `Store.getInstance().getLockingString()`) before our normal field write advice is invoked (as described in Section 4.1.1). The @singleThreaded lock is released by the `after()` advice which is invoked after our normal field write advice. The specification of initialization field read advice is similar.

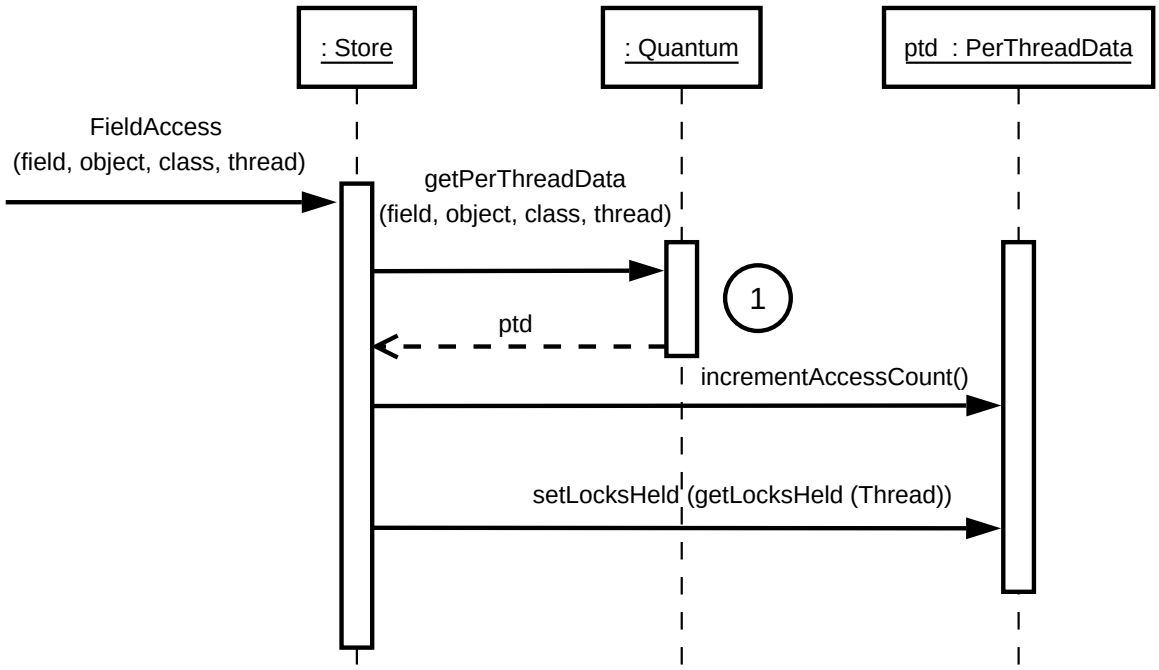
to the instrumentation portion of our tool implementation. Our rationale for this decision is that AspectJ is an evolving language and far less stable than Java. This design decision also ensures that we can change our technical approach to FlashLight instrumentation (thereby removing our dependency on AspectJ) with little impact on the rest of the implementation. We also note that the tools for developing and debugging standard Java are, currently, far superior to AspectJ. Limiting, as much as possible, the amount of AspectJ code within the FlashLight tool improves our tool design with respect to future flexibility.

An important design consideration of the FlashLight data store was to properly protect its contents from concurrent access. Therefore, we documented and verified the data store's locking policy using the Fluid assurance tool.

4.2.1 Instrumentation–Store Interaction. This section describes the interaction between the instrumentation and the data store using a series of UML sequence diagrams. These sequence diagrams provide examples of how data is collected about

the running program. The instrumentation "triggers" the collection and is responsible to extract the "raw" data from the running program. The instrumentation then sends the raw data into the data store. The data store is responsible for storing and organizing the data.

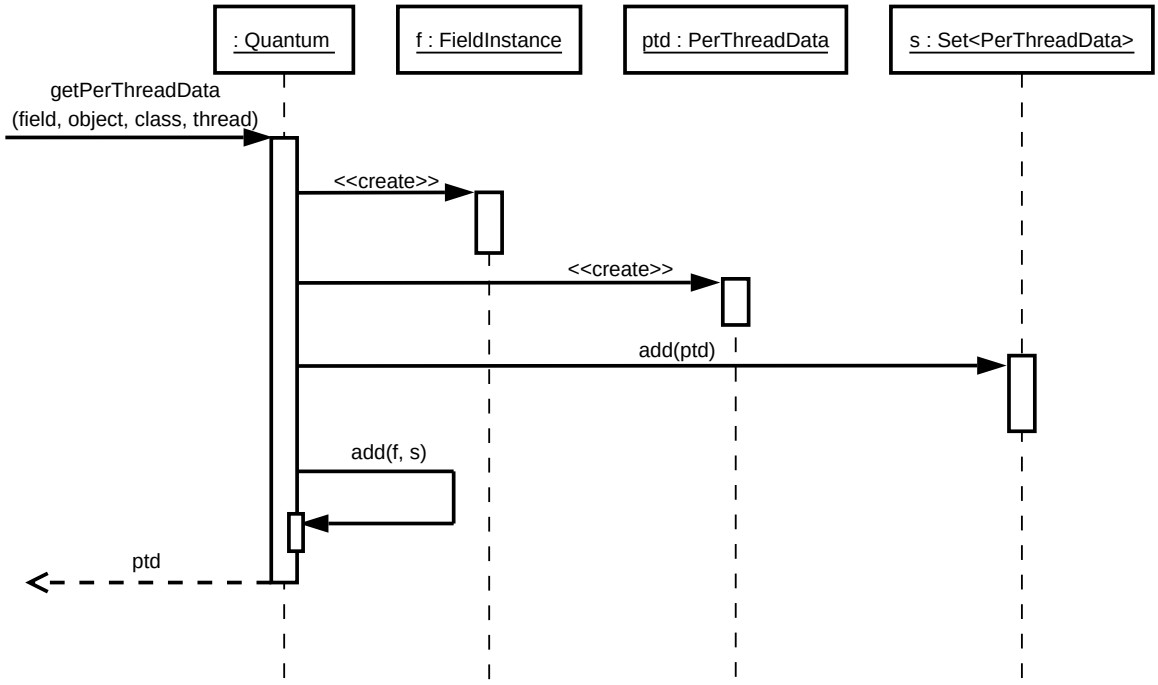
The first sequence diagram shows the dynamic interaction of objects in the data store as they record a field access triggered by our AspectJ instrumentation. Here we combine reads and writes into *accesses*. ① elides the interaction required to obtain (and possibly create) the correct `PerThreadData` object for the state accessed. This interaction is detailed in the next sequence diagram. The `PerThreadData` object contains all the data FlashLight collects about a piece of state per thread.



The `PerThreadData` object has its read or write count incremented (depending upon the type of access the instrumentation detected) and is informed of the locks held by the thread when the access occurred.

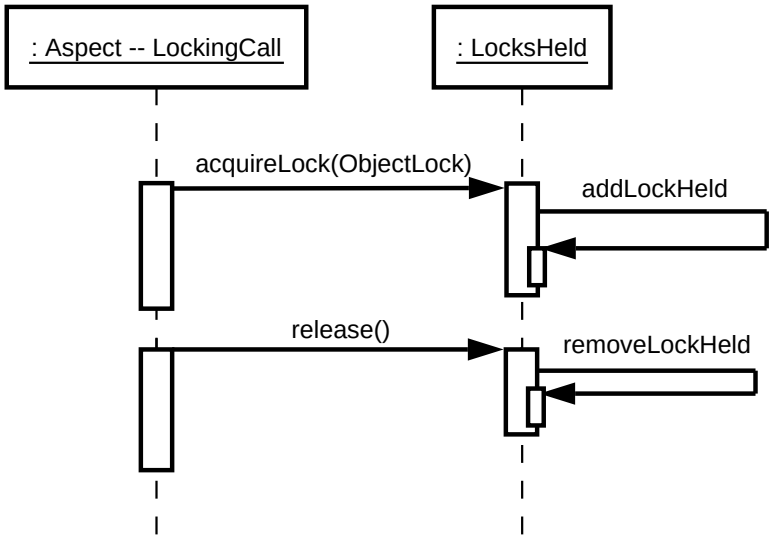
The next sequence diagram shows the first access of a field by any thread. A `FieldInstance` object is created to identify, to the data store, a particular piece of state (i.e., a field within a particular object instance). A `PerThreadData` object is

constructed to record the number of reads and writes of this state by one thread. The `PerThreadData` object contains all the data FlashLight collects about a piece of state per thread.



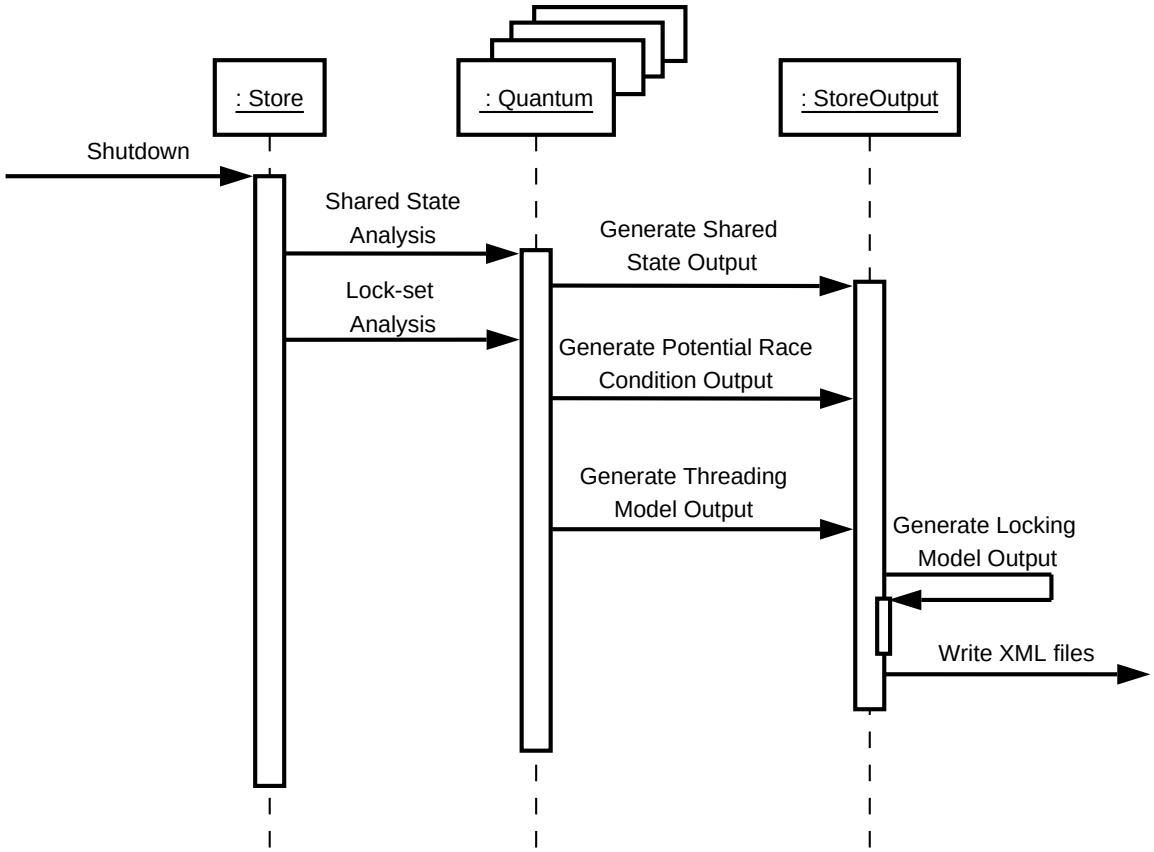
A map from `FieldInstance` objects to a set of `PerThreadData` objects (one per thread which accesses the field) is maintained by the quantum. This interaction results in a reference to the correct `PerThreadData` object, `ptd`, being returned to the caller.

The sequence diagram below shows how the data store tracks lock acquisitions and releases by threads. The instrumentation calls the `acquireLock` method on the singleton `LocksHeld` object. This call is made by the thread acquiring the lock, so by obtaining the current thread, the data store is able record the lock acquisition for the correct thread.



The instrumentation calls the `release` method to inform the data store that the lock has been released. The `LocksHeld` class maintains a list of locks currently held by every thread in the program.

The final sequence diagram shows the steps to perform data analysis and output, for each quantum, reports for the programmer. The request to terminate FlashLight originates from the program-specific aspects. At this point, the tool stops collecting data and runs data analysis for each quantum. The shared state algorithm produces the shared state report. The lock-set algorithm produces two reports: the potential race detection report and the threading model report. The fourth report, the locking model output, is produced based on the threading model report.



Output reports take the form of XML files that are created in the `xml` folder at the root of the program's Eclipse project.

4.2.2 Object Model. Figure 4.8 shows the UML class diagram of our design for the FlashLight data store. An example UML object diagram, corresponding to Figure 4.8, is shown in Figure 4.9. This object diagram shows the store organization of the data collected on a subset of the fields from the Maze ADT example (described in Chapter I). The object diagram contains three `FieldInstance` objects: `pointList`, `c`, and `Maze_size`. We note that `pointList` and `c` represent fields of the same `Maze` object instance. The fields `pointList` and `c` are accessed by two threads *main* and *AWT-EventQueue-0*, and are mapped to sets of `PerThreadData` objects that represent these threads

We now describe the classes in Figure 4.8 using Figure 4.9 as an example.

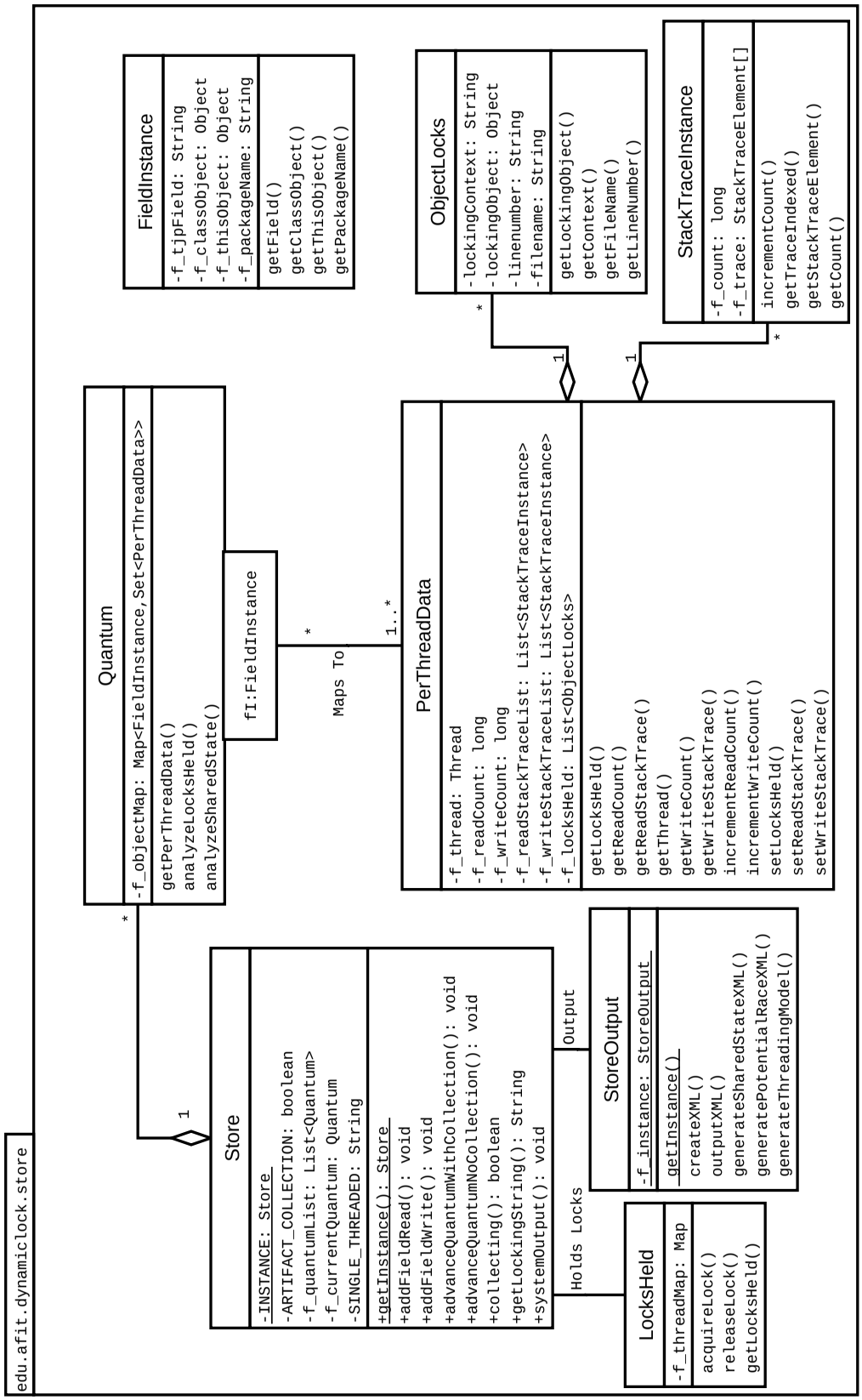


Figure 4.8: Class Diagram for the Store Package.

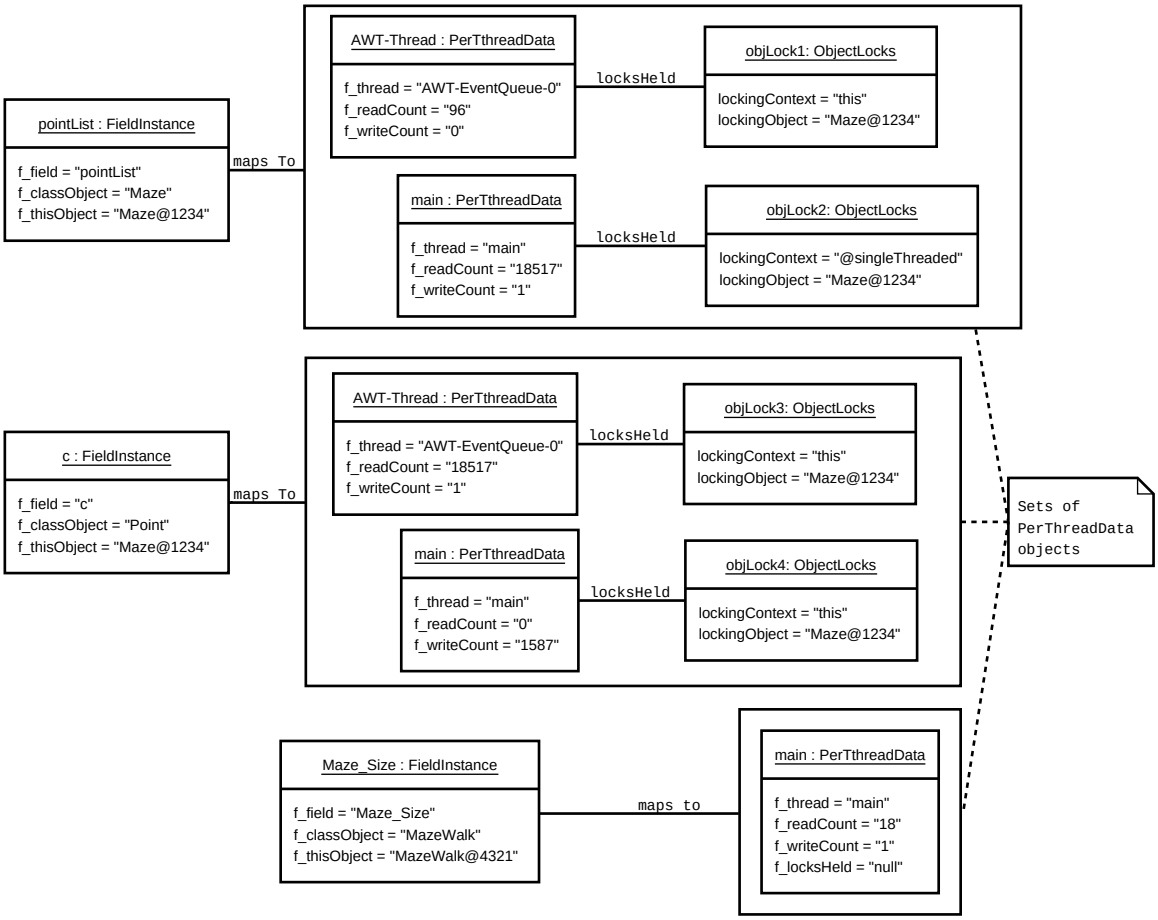


Figure 4.9: Object Diagram of the Data Store for the Maze ADT Program. The diagram shows collected data about three fields within the program: `Maze.pointList`, `Point.c`, and `MazeWalk.Maze_Size`. The Maze ADT program was described in Chapter I.

4.2.3 Store. The store class implements a *Façade* to control access to the FlashLight data store from the instrumentation. The instrumentation reports raw data to this interface. For each field access, the instrumentation records

- The object representing the *class* of the accessed field.
- The object representing the *object* of the accessed field.
- The string representing the name of the field.

As well as the following characteristics about the type of access:

- The type of access $\{\mathsf{READ}, \mathsf{WRITE}\}$.
- The thread object accessing the field.
- Any object used as a lock to protect the field access.

The store class combines the field information (class, object, field name) into a new object representing each field. These objects are called `FieldInstance` objects.

4.2.4 FieldInstance. A unique `FieldInstance` instance is created for each element of possibly shared state accessed by the program. It represents a field within an object on the program's heap. These objects are used by the data store as unique identifiers to a particular piece of state. Thus, they are typically used as the key in maps to data about the program's use of that state. For example, in Figure 4.9, the `pointList` and `c` fields map to two `PerThreadData` objects which hold information about accesses to the corresponding field by those threads.

4.2.5 Quantum. FlashLight, as described in Chapter III, allows the programmer to partition the running program into time quantums. The `Quantum` class in Figure 4.8 serves as a container for all data collected during a programmer-defined time quantum. Therefore in our design, the object diagram shown in Figure 4.9 represents the contents of a `Quantum` object.

Multiple threads can access any field, therefore a set of `PerThreadData` objects is referenced by each `FieldInstance` in the quantum's map. The map contains a record of all the fields accessed during the quantum (i.e., the `FieldInstance` objects) and records information about each field access based on the thread accessing the field (i.e. the `PerThreadData` objects).

The `Quantum` class contains a method `getPerThreadData` that returns the correct `PerThreadData` object for a given field and thread. If the given field has been accessed previously by this thread (i.e., it exists as a key in the quantum's map) then an existing `PerThreadData` object is returned, otherwise a new `PerThreadData` object is created.

4.2.6 PerThreadData. `PerThreadData` objects track every access of a field by a particular thread. The number of times a thread reads or writes a field is tracked by counters within the `PerThreadData` object. The `PerThreadData` object also keeps two lists, one for reads and one for writes, that contain stack traces documenting how the program reached a particular read or write. To limit memory consumption of FlashLight, the number of stack traces collected may be restricted by the programmer.

A `PerThreadData` object also references a list of locks held by this thread when accessing the field. Every time a thread accesses a field, the list of locks held is refined by intersecting the list of locks held at previous accesses with the locks held at the current access. The list of locks held only contains locks consistently held for all field accesses by this thread. This list is the first part of the lock-set algorithm. The analysis assumes each `PerThreadData` object maintains its own list. At each repeated field access the `PerThreadData` object contains the locks that are consistently held by this thread.

4.2.7 StackTraceInstance. A stack trace is generated for each field access. The stack trace is generated by throwing an exception and then catching it to obtain the associated stack trace array. Stack trace arrays are stored in `StackTraceInstance`

objects. Each `StackTraceInstance` object contains the trace array and the number of times it is generated by a thread. The `PerThreadData` objects compare `StackTraceInstance` objects and only store unique instances in the stack trace list.

4.2.8 ObjectLocks. The instrumentation provides additional information concerning objects used as locks. Two pieces of information are gathered about each object used as a lock: the object reference and the context of how the lock is used. The object reference allows us to identify locks in the presence of aliases. The context provides insight into how the lock is syntactically referenced in the program. For example, when a field is accessed within a `synchronized` method the context of locks protecting the access is `this` because that is the reference used to refer to the lock object.

4.2.9 StoreOutput. The `StoreOutput` class is used to report the results of the analysis. XML files are created to report results from the shared state algorithm and lock-set algorithm. Our tool output is described in Section 3.3 (on page 41).

4.2.10 LocksHeld. The `LocksHeld` class contains a mapping of threads to a list of the locks held by that thread. Thus, it is responsible for tracking the current set of locks held by each thread in the running program. Then when a field is accessed, the `Store` object requests the list of locks held by the thread accessing the field.

4.3 The Analysis

FlashLight performs several analyses based on the data store. These analyses adhere to the formalisms defined in Section 2.1. In this section we describe our shared state and lock-set algorithms, the enhancements to the lock-set algorithm we implement, and describe how the lock-set algorithm infers Greenhouse-style [8] locking models.

4.3.1 Shared State Algorithm. The shared state algorithm executes on each quantum in the store. The shared state algorithm classifies fields as *shared* when, two threads access the field and at least one access is a WRITE. For example, referring to Figure 4.9, the object diagram contains three `FieldInstances`. Each `FieldInstance` maps to a set of `PerThreadData` objects. The `PerThreadData` objects contain information about the threads accessing the fields. For two `FieldInstances`, representing the `pointList` and `c` fields, the size of the Set is greater than one. This implies more than one thread accesses this field. We also see at least one thread writes a value to each field. Based on this example, FlashLight reports the fields `pointList` and `c` as *shared*.

The shared state algorithm does not consider how fields are protected from concurrent access. We implement a lock-set algorithm to determine if fields are consistently protected.

4.3.2 Lock-Set Algorithm. The lock-set algorithm executes on each quantum in the store, just as in the shared state algorithm. The lock-set algorithm, however, evaluates the held locks by all thread for each field access. Referring to Figure 4.9, we see through the *locksHeld* association, each `PerThreadData` object maintains a list of locks consistently held while accessing its associated field. The lock-set algorithm creates a list of all locks held by all threads accessing a field. The `allLocksHeld` list is generated by adding each unique held lock by any thread accessing a field.

Recall our formalism for determining a race condition from Section 2.1.2. The lock-set algorithm iterates through the set of `PerThreadData` objects, comparing the held locks of each `PerThreadData` object against the `allLocksHeld` list. If a lock is in the `allLocksHeld` list and not in a `PerThreadData` objects held locks list, then the lock is removed from the `allLocksHeld` list because this lock in not consistently held by all threads. The lock-set determines if a field is consistently protected by iterating over the entire set of `PerThreadData` objects for a `FieldInstance`. If the `allLocksHeld` list is empty a potential race condition warning is passed to the output.

For example, using the FieldInstance `c` in Figure 4.9, we show how the lock-set algorithm determines a field is consistently protected. We construct the `allLocksHeld` list containing one object, `objLock3`. In this case, the lock-set algorithm only adds one `ObjectLock` object to the list because the objects represent the same lock. The lock-set algorithm compares the held locks for each `PerThreadData` against the `allLocksHeld` list. The lock-set algorithm produces an `allLocksHeld` containing one `ObjectLocks` object because the held locks for each `PerThreadData` object contains the lock. The results report field `c` is consistently protected by locking on the `Maze` instance.

As a rule, an empty `allLocksHeld` list implies a potential race condition. However, as stated in [20] there are common programming practices that safely access fields that violate the lock-set algorithm. Our lock-set algorithm accounts for two of these special cases.

4.3.3 Lock-Set Support for Java Programming Idioms. We discovered during our case studies that the basic lock-set algorithm reports common programming idioms as race conditions. We modify the lock-set algorithm to handle these idioms and, therefore, reduce the number of false positives reported by FlashLight.

As discussed in Section 4.1.3, the instrumentation adds a fake @singleThreaded lock to the held locks list for any thread accessing a thread during a constructor. We see in Figure 4.9 the *main* thread acquires the @singleThreaded lock when accesses `pointList`. The @singleThreaded lock allows the lock-set algorithm to distinguish between protected field accesses and constructor field accesses. The held locks for any `PerThreadData` object holding the @singleThreaded lock is not compared against the `allLockHeld` list, preventing the lock-set algorithm from reporting constructor accesses as potential races.

Consider the `pointList` FieldInstance in Figure 4.9. The `allLocksHeld` list for this field contains two `ObjectLocks` objects, `objLock1` and `objLock2`. The object `objLock2` refers to the @singleThreaded lock. The lock-set algorithm iterates over the

held locks for each `PerThreadData` object producing an `allLockHeld` list containing one object, `objLock1`. The results will report field `pointList` is consistently protected by locking on the `Maze` instance.

The modification to our lock-set algorithm reduces the number false positives reported. These fields are properly reported as being consistently protected, thus allowing the algorithm to infer a lock policy model for the fields.

4.4 Summary

This chapter presents the design and implementation of the three primary components of the FlashLight tool. The instrumentation component observes the running program and reports raw data. This raw data is organized and stored by the data store component. The organized data is then analyzed to produce output reports for the programmer.

V. Case Studies

We applied FlashLight to a number of concurrent Java programs including educational software, an established open source project, and a commercial system. Summary information on these programs is shown in the table below.

System	kSLOC	Description
FleetBaron	3	Network-based real-time strategy game
jEdit	72	A widely used open source text editor
Commercial	100	A shipping web application server

We performed the study of FleetBaron and jEdit at AFIT; we performed the commercial case study on-site with the help of the programmers that develop and maintain the system. The author did not perform the commercial case study: a committee member, Lt Col Halloran, performed this case study.

We discuss the FleetBaron case study in Section 5.1, the jEdit case study in Section 5.2, and the commercial case study in Section 5.3. In Section 5.4 we present the runtime overhead we observed when running a target program with FlashLight collecting data.

In each of our case studies, FlashLight found potential race conditions. In a few instances, such as a field in jEdit, the race condition was obvious based on inspecting the source code guided by FlashLight's output. In other examples, we were unable to determine if a real program fault existed, primarily due to our limited understanding of the program (especially in the case of jEdit and the commercial web application server). We used the jEdit case study to test the potential utility of the suggested locking models when using FlashLight as a starting point for program verification using the Fluid assurance tool. We describe, in Section 5.2.2, a case where a FlashLight proposed locking model was successfully used to verify the locking model of a jEdit class using the Fluid assurance tool.

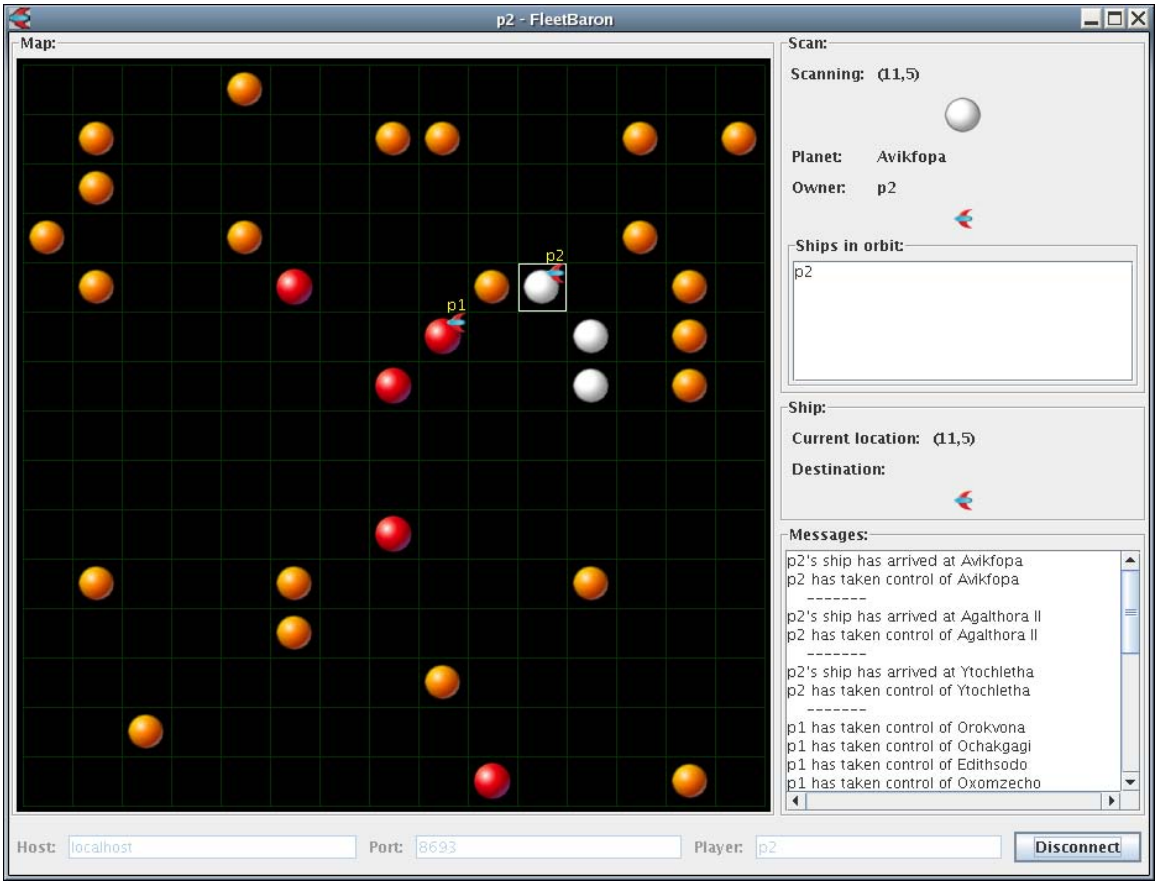


Figure 5.1: This screen shot shows a player interface from FleetBaron. This player interface shows two players, `p1` and `p2`. The planets captured by `p2` are shown in white and planets captured by `p1` are shown in red. While not shown, the FleetBaron server maintains the state of the game, coordinates interaction of the clients.

5.1 FleetBaron

The FleetBaron software is used as part of the software engineering curriculum at AFIT. FleetBaron is a concurrent, client-server based, multi-player game. The concept of the game is to "fly" your ship around the galaxy and capture as many planets as possible. The multithreaded game server creates a thread to serve each client. The server also creates threads to maintain the game state by controlling when events occur in the game. Additionally, the server coordinates all player communications. The clients communicate through sockets with the server to share the state of the game. Each client displays the game state to the user via the GUI shown in Figure 5.1

We selected FleetBaron as our first case study because of our familiarity with its design and implementation. It was primarily used as a test case for the development of FlashLight. This case study tested our concept of dynamic instrumentation via AspectJ, our ability to store collected data, our lock-set implementation, and our output reports.

5.1.1 Lessons Learned from FleetBaron. Our experience with FleetBaron exposed some areas within our early tool that needed improvement. We summarize some of our observations below.

- **Tool output.** The early output lacked any formatting. Instead, we dumped the results into a text file. The text file contained all the information about each field access, however it lacked organization making the tool output unintelligible. We modified the output to create XML files. We also constructed XSL style sheets to organize and present the information from the XML files in a clear, concise format. This improvement in the output format allowed detailed inspection of the results by all users of FlashLight.
- **What constitutes a race condition?** We observed that the analysis was reporting a high number of false positive race conditions after reviewing out-

put produced by running FleetBaron. FlashLight was reporting the majority of shared fields as potential race conditions. Investigation of these reports indicated that these reports were due to no locks being held during object construction. As described in Section 4.1.3, we refined our instrumentation and analysis to account for the common Java programming idiom of not protecting shared stated during object construction. This significantly reduced the number of false positive race conditions reported.

- **Odd locking.** Another observation from the FleetBaron case study involves a field within different object instances being consistently protected by different locks. One example of these multiple instances is shown in Figure 5.2. During one execution of FleetBaron, the server accessed the yCoordinate field of three different Location objects. For two of the Location objects, FlashLight detects the same locking policy: the lock <this>.edu.afit.fleetbaron.common.game.Ship@13582d. For the remaining instance, however, FlashLight detects that access to yCoordinate is protected by three locks:

  - <this>.edu.afit.fleetbaron.common.game.Ship@13582d
  - <@singleThreaded>.(12,15)
  - <this>.Thread[client handler p1,5,]

  This location instance is different from all other locations, because the first player's ship starts at this location. This object instance is an example of how FlashLight handles the programming idiom of single threaded constructors. By drilling down into the FlashLight's results we see why the location instance, (12,15), appears to be protected by three locks. The two write accesses performed by the `client handler p1` thread initialize the location object and add the <this>.Thread[client handler p1,5,] and <@singleThreaded>.(12,15) to held locks list. The other field accesses by the `client handler p1` thread hold these locks, and in addition they also hold the <this>.edu.afit.fleetbaron.common.game.Ship@21b6d lock. The second thread, `TurnCyclicBarrier`, ac-

- **Field yCoordinate in class edu.afit.fleetbaron.common.game.Location**

Locks consistently held by threads accessing field: **yCoordinate**

- @lock yCoordinateLOCK is <this>.edu.afit.fleetbaron.common.game.Ship@13582d protects yCoordinate

Instance	Thread Name	Read Count	Write Count
(10,15)			
	client handler p1	3	2
	TurnCyclicBarrier	4	0

Locks consistently held by threads accessing field: **yCoordinate**

- @lock yCoordinateLOCK is <this>.edu.afit.fleetbaron.common.game.Ship@13582d protects yCoordinate

Instance	Thread Name	Read Count	Write Count
(12,11)			
	client handler p1	1	2
	TurnCyclicBarrier	5	0

Locks consistently held by threads accessing field: **yCoordinate**

- @lock yCoordinateLOCK is <this>.Thread[client handler p1,5,] protects yCoordinate
- @lock yCoordinateLOCK is <@singleThreaded>.(12,15) protects yCoordinate
- @lock yCoordinateLOCK is <this>.edu.afit.fleetbaron.common.game.Ship@13582d protects yCoordinate

Instance	Thread Name	Read Count	Write Count
(12,15)			
	client handler p1	6	2
	TurnCyclicBarrier	1	0

Figure 5.2: Several proposed locking models for the `yCoordinate` field in the `Location` class. The first two accesses are consistently protected by the lock <this>.edu.afit.fleetbaron.common.game.Ship@13582d. The third instance is protected by this lock and two additional locks. In cases when more than one lock protects a field access, FLASHLIGHT reports all locks consistently held during all accesses of a field for each instance.

cesses the field once and holds a @singleThreaded lock on the (12,15) location instance. During the lock-set analysis any lists of locks containing a @singleThreaded lock as the last lock acquired are not intersected against other list of held locks. Therefore, all three locks are reported as being consistenlty held for these field accesses.

The common protection idiom is to protect a field with a single lock. Because our analysis does not account for programmer intent, FlashLight reports all locks consistently held at each field access for that instance. In the above example, all field accesses of yCoordinate not within a constructor are consistently protected by locking on the object instance *Ship@13582d*. As we discussed, the Location instance, (12,15), reports three held locks because of FlashLight's handling of the programming idiom of single threaded constructors.

5.2 jEdit

After we implemented our refinements from the FleetBaron case study, we performed another case study using the programmer's text editor, jEdit. We selected jEdit because it is a freely available, roughly 72kSLOC, production quality, Java-based multithreaded application. jEdit can be downloaded from the the project website and used with any operating system. The case study used jEdit version 4.3.

Our case study consisted of running jEdit from within Eclipse and manipulating a jEdit buffer (i.e., using the program as a text editor). We performed a *Find and Replace* operation on the buffer and replaced two strings. We selected this operation because it is multithreaded. Upon the completion of the Find and Replace operation the buffer was closed and we exited jEdit.

5.2.1 Lessons Learned from jEdit. We summarize some of our observations from using FlashLight on jEdit below.

```
1 pointcut steadyState() : call(* *..jEdit.finishStartup(..));
2
3 before() : steadyState()  {
4  Store.getInstance().advanceQuantumWithCollection("Steady State","jEdit");
5 }
```

Figure 5.3: An example of the pointcut to start data collection for jEdit. This pointcut advances the quantum after the application completes its initialization.

- **Pointcut discovery.** Determining which join points to match to advance the quantum takes reasoning about the application. The most complete results are obtained from FlashLight by using a single quantum. However, precise quantum definitions can be used to decrease the overhead introduced by FlashLight.
- **Initial pointcut.** The instrumentation provides options when to start and stop data collection by designating program specific aspects. The large size of jEdit requires attention to when to begin the data collection to reduce overhead. The jEdit startup phase includes building the GUI. The GUI contains fields that do not need to be captured or analyzed. Therefore, data collection is not started until after jEdit completes the start up phase. Figure 5.3 shows a pointcut matching a method call to `finishStartup` that indicates jEdit is done starting up. This pointcut weaves in advice to advance the quantum and start the instrumentation.
- **Termination pointcut.** Another pointcut is created to terminate collection, run the analysis, and output the results. This pointcut matches any calls to `System.exit()` from jEdit. Therefore, when *Exit* is selected from the program's GUI menu, the FlashLight analysis runs and outputs its results to the `xml` folder and then jEdit exits.
- **Running jEdit.** Running FlashLight on a project the size of jEdit was an obvious concern. Will FlashLight scale to a project this size? jEdit executed with only minimum lag while FlashLight executed. We observed that jEdit took 1.7 times longer to execute with FlashLight instrumentation compared with a non-instrumented execution. The most noticeable lags occurred with Input/Output operations, when jEdit was performing background work.

- **Compile time.** The AspectJ compiler is not as refined as the standard Java compiler. There is a noticeable difference between compiling an application with the standard Java compiler and compiling the same application with the AspectJ compiler.
- **Evaluating the output.** The output files have gone through numerous iterations to improve their presentation. In addition to the presentation, we also improved some functionality such as embedded navigation links. These links allowed a user to navigate within a file and also back to home page.

5.2.2 Verifying a jEdit Locking Model. During our jEdit case study we used the Fluid assurance tool to verify a jEdit locking model proposed by FlashLight. In this section we describe the process used our observations.

FlashLight reported that there were three shared fields within the `ReadWriteLock` class: `activeReaders`, `activeWriters`, and `writerThread`. FlashLight further reported that all three fields were consistently protected by a lock held on their enclosing instance object, i.e., `this`. FlashLight proposed three "dynamic" lock policies:

```
@lock activeReadersLock is <this>.ReadWriteLock@10e6233
       protects activeReaders
@lock activeWritersLock is <this>.ReadWriteLock@10e6233
       protects activeWriters
@lock writerThreadLock is <this>.ReadWriteLock@10e6233
       protects writerThread
```

Using the "dynamic" lock policies as a starting point we annotated the source code as shown in Figure 5.4. At line 2, we declare a region, called `RWLockRegion` that is defined to contain the three fields. At line 3, we specify that a lock on `this` protects all access to data in `RWLockRegion`.

The Fluid assurance tool did not find our model consistent with the jEdit implementation. It found 6 out of 18 field accesses were unprotected (i.e., the analysis could not verify the lock was held). Examining the unprotected field accesses we discovered that they were within methods where acquiring the lock was the callers responsibility: i.e., holding the lock was a precondition to calling the method. In Fluid,

```
1 /**
2  * @region RWLockRegion
3  * @lock rwLock is this protects RWLockRegion
4  */
5 public class ReadWriteLock {
6
7   /**
8    * @mapInto RWLockRegion
9    */
10  private int activeReaders;
11
12  /**
13   * @mapInto RWLockRegion
14   */
15  private int activeWriters;
16
17  /**
18   * @mapInto RWLockRegion
19   */
20  private Thread writerThread;
21
22  public synchronized void readLock()  {
23    if (activeReaders != 0 || allowRead())
24        ++activeReaders;
25        ...
26  }
27
28  public synchronized void readUnlock() {
29    --activeReaders;
30    ...
31  }
32
33  public synchronized void writeLock() {
34    if (allowWrite())
35    ...
36  }
37
38  public synchronized void writeUnlock()  {
39    --activeWriters;  writerThread = null;
40    ...
41  }
42
43  /**
44   * @requiresLock rwLock
45   */
46  private boolean allowRead() {
47    return (Thread.currentThread() == writerThread)
48            || (waitingWriters == 0 && activeWriters == 0);
49  }
50
51  /**
52   * @requiresLock rwLock
53   */
54  private boolean allowWrite()  {
55    return activeReaders == 0 && activeWriters == 0;
56  }
57 }
```

Figure 5.4: The elided `ReadWriteLock` class with Fluid promises added to precisely specify its lock policy: when accessing the fields `activeReaders`, `activeWriters`, and `writerThread` a lock on the object instance (i.e., `this`) must be held. The Fluid assurance tool verifies this lock policy is consistent with the code.

this is indicated by annotating these methods with a `@requireslock` annotation as seen at line 44 and 52 in Figure 5.4. With this additional piece of design intent, the Fluid assurance tool was able to verify code–model consistency.

We did find FlashLight helpful in focusing our work with the Fluid assurance tool. As seen in the example described above, a "programmer-in-the-loop" is required to develop, from the FlashLight proposal, a verifiable lock policy model. Future work may be able to lower the gap between the FlashLight output and a verifiable lock policy model.

5.3 Commercial Case Study

FlashLight was used during a commercial case study on a commercial web application server. This was a high-quality shipping product in use at hundreds of customer locations. The case study was conducted on-site at the location where the software was developed and maintained and with the assistance of the product's programming team.

The focus of the study was not to try out the FlashLight tool, however, one of the developers became very interested in trying FlashLight based upon an overview of the tool presented on the first day of the case study. This developer wanted to gain a better understanding of the concurrency within the overall "thread pool" for the application server.

Configuration of FlashLight for this case study was non-trivial because the commercial web application server could not be run from within Eclipse. In addition, the server could not be run on a Java 5 JRE, it required a specific Java 2 JRE to run correctly. Therefore, portions of the FlashLight source code had to be "back-ported" to Java 2 on-site. This process that took roughly two hours to accomplish.

It took four hours (of iterative trial and error) to produce a FlashLight instrumented version of the web application server. The application server ran as expected, but with a noticeable requirement for additional memory due to the large number

of threads the server managed. The tool output described the shared state between the hundreds of Java threads the server was managing. The first run produced over 100 MBytes of output, so the instrumentation was tuned to focus on state within particular areas of the server the programmer was interested in. This tuning reduced the size of the output. The programmer found the FlashLight output of this second run of the server informative.

Feedback we received from the programmer included

- (+) The use of quantums and the flexibility of the aspect-based instrumentation to tune FlashLight to the target program was considered beneficial. The programmer reported that other (unnamed) dynamic analysis tools had not been able to support analysis of the commercial web application server FlashLight successfully analyzed.
- (-) The FlashLight output for the first run was very slow to render in a web browser. Taking up to 4 minutes to appear. The programmer, who had spent two days using the Eclipse-based Fluid assurance tool also wanted to view the FlashLight results within Eclipse (not using a web browser).
- (-) The slowest portion of the trial and error tuning of FlashLight to the web application server was the speed of the AspectJ compiler. After adjusting the definition of an aspect (e.g., to define a quantum or trigger analysis and output) it took 5 minutes on the laptops being used for the case study to run the AspectJ compiler over the web application server.

Overall, the programming team of the web application server saw FlashLight as useful and expressed an interest in further development of the tool (including addressing the (-) drawbacks listed above). FlashLight had been successful in their environment where previous dynamic tools they had tried had failed.

Table 5.1: This table describes the run-time performance of three programs tested with FlashLight. The unmodified column reports the amount of wall clock time (in seconds) required to execute the programs without any instrumentation. The "full execution" column reports the elapsed time when FlashLight is "on" for the entire program duration. The last column reports the elapsed time FlashLight actively collects data for the system. FlashLight's instrumentation divides the system into multiple quantum, with some quantum not collecting any data.

System	Unmodified	Full Execution	Quantized Execution[a]
FleetBaron PlayerUI	37s	55s	47s
FleetBaron Server	51s	69s	61s
jEdit	46s	150s	79s

[a]Quantized Execution implies program executions is broken into multiple quantum, and assumes some quantum do not capturing field accesses.

5.4 Runtime Overhead

This section characterizes, based upon our use, the runtime overhead introduced by FlashLight. The dynamic weaving of FlashLight's instrumentation affects the program's execution. What are the significant factors affecting the increase of system requirements when running FlashLight and how much does FlashLight affect a program?

The FleetBaron and jEdit case studies were run on an IBM laptop with a 1.6GHz Pentium 4 processor and with 1GB of memory. We used the Eclipse IDE with the AspectJ plug-in and a Java 5 JRE.

The runtime overhead introduced by FlashLight on three programs is reported in Table 5.1. Let us review our jEdit test plan. Because of the GUI driven commands of jEdit, we developed a test plan allowing us to consistently evaluate the tool from opening jEdit until termination of the application. The plan consisted of opening a file, performing a search and replace command, closing the file, and exiting jEdit. Both operations, the open and close file commands and the search and replace command, allow FlashLight to capture concurrent field accesses. Admittedly, we could achieve more accurate results using an automated tool to perform our test plan, however, due to time constraints, we performed the test plan manually to provide baseline results.

Referring to Table 5.1, we see that executing our test plan with an unmodified version of jEdit took approximately 46 seconds. During our case study, with FlashLight running, jEdit took approximately 79 seconds to execute the application, analysis and output the results. As we have discussed, separating an applications into different quantum can reduce the overhead incurred from FlashLight. We see there is possible time savings in using multiple quantums by comparing the full and quantized executions in Table 5.1. We assume, however, the risk of also reducing the accuracy of the analysis.

There are several scalability challenges for FlashLight. The size of an application (i.e. kSLOC) is not the only factor in determining the overhead incurred by FlashLight. Although jEdit is considerably larger than the FleetBaron server, there is little difference between the quantized execution times of the systems (Table 5.1). The size of program (i.e. number of lines of code) is not the sole factor in determining a programmer's overhead. A system's size, the number of fields, and the number of field accesses are all significant factors in determining the overhead added by FlashLight.

5.5 Summary

Our case studies provide initial evidence that FlashLight is scalable (up to 100kSLOC), is effective in finding race conditions, and assists programmers by providing suggested lock policy models. The case studies also demonstrated some deficiencies in our early implementation, namely, the format of the results

- The *effectiveness* of FlashLight was shown in each case study by finding real race conditions, and suggesting potential lock policy models.
  - Faults: Discovered an actual race conditions in jEdit. We realized it took some time to transition from classifying a field as a potential race, to using Fluid to show that it was in fact a race condition.
  - False Positives: By enhancing the FlashLight lock-set algorithm we reduced the number of false positives reported by the tool. Cutting the

number of reported races in the FleetBaron server from ~ 30 fields to 5 fields.

- Our initial output implementation did not provide clear, concise information degrading the *user experience*. Through several iterations of the output, we now report summarized, relevant results. Users can view detail information by drilling-down into the results using built-in navigation links.
- Our case studies showed FlashLight is capable of working on large applications. This *scalability* ensures FlashLight can be used on a wide range of applications. We used FlashLight on applications up to 100kSLOC, however this is not a firm boundary. The upper bound of the tool appears to be how long a programmer wants to wait for the AspectJ compiler. For example, during the commercial case study the AspectJ compiler took several minutes to compiler code while Eclipse complied the code in under a minute.
- The case studies also demonstrated FlashLight's *practicality*. FlashLight was used in one commercial, on-site case study conducted by a fellow researcher with professional programmers. This team focused on the using the Shared State and Threading Model views generated by FlashLight. The case study team was excited about FlashLight's tunability from AspectJ and flexibility because unlike other dynamic tools, FlashLight executed within their environment, an application server cluster.

VI. Conclusion

Reasoning about Java concurrency is not easy. A lack of understanding of the concurrency within a system can lead to race conditions and deadlocks. These errors are difficult to locate and correct. Our dynamic analysis tool, FlashLight, provides one link in a chain of programmer-oriented tools to safe concurrency. FlashLight locates shared state, potential race conditions, and suggests possible locking models from the run-time environment of a program. The suggested locking models can be used with the Fluid Lock Assurance to assure the code. This combination of dynamic and static analysis tools creates a powerful toolset for illuminating developers about potential errors and verifying their code.

6.1 Summary of Contributions

This thesis describes a dynamic analysis tool, named FlashLight, that detects shared state and potential race conditions within a program. The tool, based upon a program's observed locking behavior, also proposes Greenhouse-style [8] lock policy models that can, after review by a programmer to ensure reasonableness, be assured by the Fluid assurance tool. Overall, FlashLight is designed and implemented to help "shed some light" on a programmer's understanding of the concurrency in a Java program. It has also been designed to be synergistic with the Fluid assurance tool—toward the goal of improving the quality of large real-world software system in a practical manner.

The combination of a dynamic tool with a program verification system focused on concurrency fault detection and repair is, to the best of our knowledge, novel and is the primary contribution of this research. A secondary contribution of the work is the extension of the lock-set analysis algorithm to use *quantums*. Quantums allow the programmer to specify one or more "interesting" periods of time during a program's execution.

6.1.1 Case Studies. We applied FlashLight to a several concurrent Java programs including educational software, an established open source project, and a commercial system. Our case studies highlighted several opportunities to improve FlashLight, such as reducing the number of false positives reported by tuning the lock-set algorithm used by the tool to support typical Java programming idioms. As part of our case study, we evaluated the overhead incurred by using FlashLight. During our trials, the open source text editor jEdit took approximately 1.7 times longer to execute while being inspected with FlashLight. Our case studies also pointed out the necessity to revise our output presentation. Significant work was required to make the outputted web pages understandable and useful. Our case studies highlighted several serious flaws in our early tool output.

6.2 Looking Ahead

We propose the following improvements to the FlashLight tool:

1. Integrate tool output directly into Eclipse and avoid the intermediate browser output. This would increase the usability of the tool by making it easier for the user to see the results in one view opposed to several views.
2. Support better integration with Fluid. Currently, there are only two Fluid annotations used in the output. The instrumentation could be expanded to collect more data and allow the analysis to infer more about the developers intent. In the special case where multiple locks consistently protect a field, determining which lock is required to protect this field access.

FlashLight illuminates developers on the concurrency within their system. Using FlashLight in conjunction with the Fluid assurance tool creates a powerful and practical quality assurance technique aimed at consistently producing better concurrent Java code.

Bibliography

1. Bierhoff, Kevin and Jonathan Aldrich. "Lightweight object specification with typestates". *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, 217–226. ACM Press, New York, NY, USA, 2005.

2. Boroday, S., A. Petrenko, J. Singh, and H. Hallal. "Dynamic analysis of java applications for multithreaded antipatterns". *WODA '05: Proceedings of the third international workshop on Dynamic analysis*, 1–7. ACM Press, New York, NY, USA, 2005.

3. Choi, Jong-Deok, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. "Efficient and precise datarace detection for multithreaded object-oriented programs". *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, 258–269. ACM Press, New York, NY, USA, 2002.

4. Christiaens, Mark and Koen De Bosschere. "TRaDe, A Topological Approach to On-the-fly Race Detection in Java Programs". *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM'01)*. USENIX, 1991.

5. Engler, Dawson and Ken Ashcraft. "RacerX: effective, static detection of race conditions and deadlocks". *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, 237–252. ACM Press, 2003.

6. Goldberg, Allen and Klaus Havelund. "Instrumentation of Java Bytecode for Runtime Analysis". *FTfJP'03, Fifth ECOOP Workshop on Formal Techniques for Java-like Programs*. 2003.

7. Gosling, James, Bill Joy, Guy L. Steele, and Gilad Bracha. *Java Language Specification*. Addison-Wesley, 3nd edition, 2005.

8. Greenhouse, Aaron. *A Programmer-Orientated Approach to Safe Concurrency*. Ph.D. thesis, Carnegie Mellon University, May 2003.

9. Greenhouse, Aaron and William L. Scherlis. "Assuring and Evolving Concurrent Programs: Annotations and Policy". *24th International Conference on Software Engineering (ICSE'02)*, 453–463. ACM Press, May 2002.

10. Havelund, Klaus. "Using Runtime Analysis to Guide Model Checking of Java Programs". *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, 245–264. Springer-Verlag, London, UK, 2000.

11. Holzmann, Gerald. *The Spin Model Checker – Primer and Reference Manual*. Addison-Wesley, New York, 2004.

12. Kiczales, Gregor, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. "An Overview of AspectJ". *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, 327–353. Springer-Verlag, London, UK, 2001.

13. Kiczales, Gregor, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. "Aspect-Oriented Programming". *ECOOP '97: Proceedings of the 11th European Conference on Object-Oriented Programming*, 220–242. Springer-Verlag, 1997.

14. Lamport, Leslie. "Time, clocks, and the ordering of events in a distributed system". *Comm. of the ACM*, 21(7):558–565, 1978.

15. Lea, Doug. *Concurrent Programming in Java*. The Java Series. Addison-Wesley, Reading, Massachusetts, USA, second edition, November 1999.

16. Naumovich, Gleb, George S. Avrunin, and Lori A. Clarke. "An efficient algorithm for computing MHP information for concurrent Java programs". *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, 338–354. Springer-Verlag, London, UK, 1999.

17. O'Callahan, Robert and Jong-Deok Choi. "Hybrid dynamic data race detection". *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, 167–178. ACM Press, New York, NY, USA, 2003.

18. von Praun, Christoph and Thomas R. Gross. "Object race detection". *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, 70–82. ACM Press, New York, NY, USA, 2001.

19. von Praun, Christoph and Thomas R. Gross. "Static conflict analysis for multi-threaded object-oriented programs". *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, 115–128. ACM Press, New York, NY, USA, 2003.

20. Savage, Stefan, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. "Eraser: a dynamic data race detector for multi-threaded programs". *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, 27–37. ACM Press, 1997.

21. Schonberg, Edith. "On-the-fly detection of access anomalies". *SIGPLAN Not.*, 39(4):313–327, 2004.

22. Sestoft, Peter. *Java Precisely*. The MIT Press, 2nd edition, 2005.

23. Visser, Willem, Klaus Havelund, Guillaume Brat, and SeungJoon Park. "Model Checking Programs". *ASE '00: Proceedings of the 15th IEEE international con-*

ference on Automated software engineering, 3. IEEE Computer Society, Washington, DC, USA, 2000.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
23–03–2006	Master's Thesis	Jun 2004 — Mar 2006

4. TITLE AND SUBTITLE

FlashLight:
A Dynamic Detector of Shared State, Race Conditions,
and Locking Models in Concurrent Java Programs

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Hale, Scott, C., Captain, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology (AFIT/EN)
Graduate School of Engineering and Management
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCS/ENG/06-08

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Mr. William B. Martin
Chief, High Confidence Software and Systems Division Information Assurance Research Group National Security Agency
9800 Savage Road e-mail: wbmarti@alpha.ncsc.mil
Fort George G. Meade, MD 20755-6511 comm: (301) 688-1057

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT Concurrent Java programs are difficult to understand and implement correctly. This difficultly leads to code faults that are the source of many real-world reliability and security problems. Many factors contribute to concurrency faults in Java code; for example, programmers may not understand Java language semantics or, when using a Java library or framework, may not understand that their resulting program is concurrent.

This thesis describes a dynamic analysis tool, named FlashLight, that detects shared state and possible race conditions within a program. FlashLight illuminates the concurrency within a program for programmers that are wholly or partially "in the dark" about their software's concurrency. FlashLight also works in concert with the Fluid assurance tool to, based upon a program's observed locking behavior, propose lock policy models. After review by a programmer to ensure reasonableness, these models can be verified by the Fluid assurance tool. Our combination of a dynamic tool with a program verification system focused on concurrency fault detection and repair is, to the best of our knowledge, novel and is the primary contribution of this research.

15. SUBJECT TERMS

Java Programming Language, Concurrency, Dynamic Analysis, Race Condition Detection, Software Engineering, Software Tools, Computer Programming and Software

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Maj Robert P. Graham, Jr., (AFIT/ENG)
U	U	U	UU	100	19b. TELEPHONE NUMBER *(include area code)* 937-255-3636, ext. 7256 (robert.graham@afit.edu)